RealView® Developer Kit

Version 2.2

Command Line Reference Guide

ARM®

ARM DUI 0284C

RealView Developer Kit
Command Line Reference Guide

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
April 2005	B	Release for RVDK v2.2
January 2006	C	Release for RVDK v2.2 with Trace

Proprietary Notice

Confidentiality Status

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
RealView Developer Kit Command Line Reference Guide

Preface

This preface introduces the *RealView® Developer Kit (RVDK) v2.2 Command Line Reference Guide*. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page ix.

About this book

You can control the RealView Debugger by using either its *Graphical User Interface* (GUI) or its *Command-Line Interface* (CLI). This book describes the RealView Debugger CLI.

Intended audience

This book has been written for developers who are using RealView Debugger to debug software written to run on ARM® architecture-based target systems. It assumes that you are a software developer who is familiar with command-line tools. It does not assume that you are familiar with RealView Debugger.

Using this book

This book is organized into the following parts and chapters:

Chapter 1 *Working with the CLI*

Read this chapter for an introduction to the RealView Debugger CLI.

Chapter 2 *RealView Debugger Commands*

Read this chapter for a detailed description of the RealView Debugger CLI commands.

Chapter 3 *RealView Debugger Predefined Macros*

Read this chapter for a detailed description of the RealView Debugger predefined macros.

Chapter 4 *RealView Debugger Keywords*

Read this chapter for a detailed description of the RealView Debugger keywords.

Glossary See this for explanations of terms used in this book.

Typographical conventions

The following typographical conventions are used in this book:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
`monospace` (mono underlined)	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.
`monospace italic`	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See the Documentation area of `http://www.arm.com` for current errata sheets, addenda, and the ARM Frequently Asked Questions list.

ARM publications

This book is part of the RealView Developer Kit documentation suite. Other books in this suite include:

- *RealView Developer Kit v2.2 Debugger User Guide* (ARM DUI 0281)
- *RealView Developer Kit v2.2 Compiler and Libraries Guide* (ARM DUI 0282)
- *RealView Developer Kit v2.2 Assembler Guide* (ARM DUI 0283)
- *RealView Developer Kit v2.2 Linker and Utilities Guide* (ARM DUI 0285)
- *RealView Developer Kit v2.2 Project Management Guide* (ARM DUI 0291)
- *RealView Developer Kit v2.2 Extensions User Guide* (ARM DUI 0335)

For general information on software interfaces and standards supported by ARM, see *`install_directory`*`\RVDK\Examples\...`.

See the following documentation for information relating to the ARM debug interfaces suitable for use with RealView Debugger:

- *RealView ICE User Guide* (ARM DUI 0155)
- *RealView ICE Micro Edition User Guide* (ARM DUI 0220)

See the datasheet or Technical Reference Manual for your hardware.

Other publications

For a comprehensive introduction to ARM architecture see:

Steve Furber, *ARM System-on-Chip Architecture, Second Edition*, 2000, Addison Wesley, ISBN 0-201-67519-6.

For a detailed introduction to regular expressions, as used in the RealView Debugger search and pattern matching tools, see:

Jeffrey E. F. Friedl, *Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools*, 1997, O'Reilly & Associates, Inc. ISBN 1-56592-257-3.

For the definitive guide to the C programming language, on which the RealView Debugger macro and expression language is based, see:

Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language, second edition*, 1989, Prentice-Hall, ISBN 0-13-110362-8.

For more information about IEEE Std. 1149.1 (JTAG), see:

IEEE Standard Test Access Port and Boundary Scan Architecture (IEEE Std. 1149.1), available from the IEEE (`www.ieee.org`).

Feedback

ARM Limited welcomes feedback on both RealView Debugger and its documentation.

Feedback on RealView Debugger

If you have any problems with RealView Debugger, submit a Software Problem Report:

1. Select **Help → Send a Problem Report...** from the RealView Debugger main menu.
2. Complete all sections of the Software Problem Report.
3. To get a rapid and useful response, give:
   - a small standalone sample of code that reproduces the problem, if applicable
   - a clear explanation of what you expected to happen, and what actually happened
   - the commands you used, including any command-line options
   - sample output illustrating the problem.
4. Email the report to your supplier.

Feedback on this book

If you have any comments on this book, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are welcome.

Chapter 1
Working with the CLI

This chapter introduces the RealView® Debugger *Command-Line Interface* (CLI). It contains the following sections:

1.1 General command language syntax

This section describes the general syntax conventions that are supported by the RealView Debugger CLI:

- *General syntax rules*
- *Command qualifiers and flags*
- *Command parameters* on page 1-3
- *Abbreviations* on page 1-4.

1.1.1 General syntax rules

The commands you submit to the debugger must conform to the following rules:

- Each command line can contain only one debugger command.
- You can use any combination of uppercase and lowercase letters in commands.
- A command line can be up to 4095 characters in length.

1.1.2 Command qualifiers and flags

Many commands accept flags, qualifiers, and parameters, using the following syntax:

```
COMMAND [,qualifier | /flag] [parameter]...
```

If a command qualifier is present, it must appear after the command name and before any command parameters.

Qualifiers

You introduce each command qualifier with a punctuation character, as follows:

`,qualifier` A comma introduces a qualifier that provides the debugger with additional information on how to execute a command. For example, the command:

```
DHELP,FULL =command_name
```

displays the full version instead of the summary version of its help text.

Other comma qualifiers are included in the command descriptions described in Chapter 2 *RealView Debugger Commands*.

Flags

You introduce each command flag with a forward slash character, as follows:

`/flag`	A flag is either one or two letters that acts as a switch.

For example, some commands accept a size flag. Valid size flags are:

`/B`	8 bits. Sets the size of some value or values to a byte.
`/H`	16 bits. Sets the size of some value or values to a halfword.
`/W`	32 bits. Sets the size of some value or values to a word.

For an example of a command that accepts these qualifiers, see `FILL` on page 2-161.

Where a command supports flags, the flags are described as part of the command syntax (see Chapter 2 *RealView Debugger Commands*).

1.1.3 Command parameters

As described in *Command qualifiers and flags* on page 1-2, commands accept flags, qualifiers, and parameters.

Command parameters are typically expressions that represent values or addresses to be used by a command. Parameters must be separated from each other with some form of punctuation. However, punctuation for the first parameter might be optional:

`=text` An equals sign introduces a text string when you have multiple parameters. It is not required for the first parameter. Depending on the command, this might specify:

- a resource
- a thread or process name
- a number or string expression
- an address or offset
- a description
- an instance
- a location
- a configuration.

`;windowid | ;fileid`

A semicolon introduces a specification of where any output produced by the command is to be sent. If you supply a `;windowid` or `;fileid` parameter, it must be the final parameter of the command. See *Window and file numbers* on page 1-5 for details.

`;macro-call` A semicolon also introduces a specification of a macro to be called by the command. If you supply a `;macro-call` parameter, it must be the final parameter of the command. You cannot use a `;windowid` or `;fileid` parameter with a `;macro-call` parameter. If you want to send the output from the macro to a window or file, use the `VMACRO` command (see *VMACRO* on page 2-341).

Rules for specifying command parameters

The parameters you supply to a RealView Debugger command must conform to the following rules:

- One or more spaces must separate command parameters from a command when there is no punctuation (for example, a `/`, `,`, or `=`).
- If a parameter, for example a filename, includes spaces or other special characters, you must enclose it in double quotes (`"..."`), or single quotes (`'...'`).
- In high-level mode, code addresses must be referenced by line numbers, labels, and function names, or casted values.

1.1.4 Abbreviations

You can enter many debugger commands in an abbreviated form. The debugger requires enough letters to uniquely identify the command you enter.

Many commands also have aliases. An alias is a different name that you can use instead of the listed name (see `ALIAS` on page 2-25). If you can use a short form of an alias, the underlined characters show the shortest acceptable form, for example:

`BREAKI`	Is an acceptable short form of `BREAKINSTRUCTION`.
`BINSTRUCTION`	Is an alias of `BREAKINSTRUCTION`.
`BI`	Is the shortest form of `BREAKINSTRUCTION`.
`DCOM`	Is an acceptable short form of `DCOMMANDS`.
`DHELP`	Is an alias of `DCOMMANDS`.

To see if a particular CLI command has an acceptable short form or alias, see its description in Chapter 2 *RealView Debugger Commands*.

1.2 Window and file numbers

Many commands and macros enable you to specify a window number (*`windowid`*) or file number (*`fileid`*). The number identifies a window or file to which any output is sent. You must use one of the following numbers:

`1`	Code window, and journal file if enabled.
`20`	Standard I/O window. The `STDIOLOG` command described on page 2-271 also enables you to write Standard I/O messages to a file.
`28`	Log file. For more information see `LOG` on page 2-193.
`29`	Journal file. For more information see `JOURNAL` on page 2-184.
`50-1024`	User defined window or file number. For more information see `FOPEN` on page 2-166 and `VOPEN` on page 2-343.

Note

When a user-defined number is in use for a window or a file, that number cannot be reused until you close the associated window or file. Use the `VCLOSE` command to close both windows and files. For more information see `VCLOSE` on page 2-337.

To see what windows and files you have opened, use the `WINDOW` command, see `WINDOW` on page 2-350.

1.3 Using expressions and statements

The basic components of the RealView Debugger command-line language can be classified as either expressions or statements, or a combination of both, where statements are typically contained in include files.

This section describes:

- *Expressions*
- *Keywords*
- *Predefined macros*.

1.3.1 Expressions

There are many types of expressions accepted by the RealView Debugger CLI, enabling you to extend the operation of a command from the CLI. Expressions can be, for example, binary mathematical expressions, references to module names, or calls to functions. For an example of these and other types of expressions, see *Types of RealView Debugger expressions* on page 1-10.

1.3.2 Keywords

The RealView Debugger keywords are statements that can be used in a macro definition. These keywords are the same as the C language keywords, and they cannot be redefined or used in any other context. For a detailed description of these keywords, see Chapter 4 *RealView Debugger Keywords*.

1.3.3 Predefined macros

RealView Debugger also provides predefined macros. Predefined macros can be used in macros that you define, and directly at the command line when used with the `CEXPRESSION` command. For a detailed description of the predefined macros available in RealView Debugger, see Chapter 3 *RealView Debugger Predefined Macros*.

For details on creating macros and using them with RealView Debugger, see the chapter that describes working with macros in the *RealView Developer Kit v2.2 Debugger User Guide.*

1.4 Macro language

Macros are constructed in a Kernighan and Ritchie C-like scripting language that is interpreted on the host. You can create your own or use one of the available predefined run-time macros (see Chapter 3 *RealView Debugger Predefined Macros*).

This section covers:

- *Macro definition*
- *Macro body* on page 1-8
- *Macro terminator* on page 1-9
- *Macro comments* on page 1-9
- *Macro local symbols* on page 1-9.

1.4.1 Macro definition

A macro definition must contain:

- the `DEFINE` command (see `DEFINE` on page 2-110)
- the macro name
- the macro body
- a terminating full stop or period (`.`) as the first and only character on the last line.

The syntax of a macro definition is as follows:

```
DEFINE [/R] [return_type] macro_name([parameter_list])
[param_definitions]
{
    macro_body
}
.
```

where:

`/R`	The new macro can replace an existing symbol with the same name.
`return_type`	The return type for the macro and is an optional component of the macro definition. The default type is **`int`**.

Note

One use of a macro return value is to control what action RealView Debugger takes when a conditional breakpoint is triggered (for example, see *BREAKINSTRUCTION* on page 2-61).

parameter_list	A parameter list for the macro and is an optional component of the macro definition. You specify a parameter list in the same way that you specify arguments for a C function. If *parameter_list* is defined then the type must also be specified or else type **int** is assumed. The following example illustrates the use of a *parameter_list*:

```
define int scpy(target, source)
char *target;
char *source;
```

The declaration defines arguments for the macro `scpy()`. The type of both the target and the source are declared to be pointers to a `char`.

Note

For full details on the `DEFINE` command, see `DEFINE` on page 2-110.

1.4.2 Macro body

The macro body consists of the source lines of the macro and optional formal arguments.

The syntax of a macro body is as follows:

```
[local_definitions]
macro_statement;[macro_statement;]...
```

where:

local_definitions	defines variables used locally in the macro body.

Formal arguments can be used throughout the macro body. These arguments are later replaced by the values of the actual arguments in the macro call.

You can use debugger commands in the macro body. If used, you must enclose the command with dollar signs ($) and end in a semi-colon (`;`), for example:

```
last_time = @cycle;
    value = base[offset];
    base[offset] = 0;
$printf “base offset value=%d\n”,value$;
```

You can also use macro arguments and local variables in RealView Debugger commands.

1.4.3 Macro terminator

A macro terminator is used as the last character in the macro. This is a full-stop or period (`.`) and is the first and only character on the last line.

1.4.4 Macro comments

You can use comments in your macros to document your code. Any characters identified as belonging to a comment are ignored by RealView Debugger. The following rules apply to comments in macros:

- C style comments begin with a slash followed by an asterisk (`/*`) and end with an asterisk followed by a slash (`*/`). Also the comment text must be on a single line, for example:

  ```
  /* comment */
  /*
     another comment
  */
  ```

- C++ style comments begin with two slashes (`//`) and end when the end of the line is reached, for example:

  ```
  // This is a line comment
  // Copyright (c) ARM Limited
  ```

- Comments can begin with `//#` and end when the end of the line is reached.
- Only `//` or `//#` comments can be placed at the end of a macro statement, for example:

  ```
  macro_statement; // comment
  ```

- Comments cannot be nested.

1.4.5 Macro local symbols

You can create symbols in a macro that are local to the macro. You must declare a type for macro local symbols. The type can be any legal C or C++ data type. All symbols declared within macros are automatic variables, that is the `static` keyword is not recognized. To create the equivalent of a global static variable, use the `ADD` command to create the symbol before executing the macro that references the symbol. For example:

```
add int cnt
define /R counter()
{
    cnt = cnt + 1;
}
.
```

1.5 Types of RealView Debugger expressions

As described in *General command language syntax* on page 1-2, the CLI requires that you enter commands in a form acceptable to the debugger. The components of these commands are expressions and statements. For more details on statements, see *Using expressions and statements* on page 1-6.

Table 1-1 shows many of the types of expressions accepted by the CLI. For each type, there is a cross-reference to a command in Chapter 2 *RealView Debugger Commands* where the syntax accepts that expression type.

Table 1-1 Types of CLI expressions

Type	Pattern/example	Usage cross-reference
Arithmetical operation (value or address)	`x+(y*5)`	`FILL` on page 2-161
Array element reference (value or address)	`argv[1]`	`ARGUMENTS` on page 2-34
Conditional expression	`c==3`	`BREAKINSTRUCTION` on page 2-61
Floating-point expression	`3.14`	`FPRINTF` on page 2-169
Function name reference (code address)	`main`	`LIST` on page 2-187
Line reference (code address)	`#`*`line_number`*	`SCOPE` on page 2-252
Macro call	*`macro`*`()`	`ALIAS` on page 2-25
Memory address	`&`*`address`*	`ADD` on page 2-18
Memory address	`(type *)` *`expression`*	-
Memory location	`0x8000`	`BREAKREAD` on page 2-69
Memory range expression	`0x100..0x200`	`BREAKREAD` on page 2-69
Qualified line (specifying source module)	`MODULE1\#12`	`SCOPE` on page 2-252
Stack level reference	*`stack_level`*	`SCOPE` on page 2-252
String expression	`"`*`string`*`"`	`FILL` on page 2-161
Symbol reference (value or address)	*`symbol_name`*	`ADD` on page 2-18
Target connection reference	*`targetid targetname`*	`CONNECT` on page 2-98
Target program function	*`function`*`()`	`BREAKINSTRUCTION` on page 2-61

1.6 Constructing expressions

The debugger groups expressions into two classes:

- C source language expressions, used in assembled or compiled source mode
- assembly language expressions, used in assembly source or disassembly mode.

Most valid C expressions are also valid in the debugger (see *Using expressions and statements* on page 1-6). However, if you are an assembly language user, you do not have to know how to program in C to use the debugger. Simple C expressions are the same as standard algebraic expressions.

The types of expression elements accepted by the CLI are described in *Types of RealView Debugger expressions* on page 1-10. This section introduces the basic elements of the CLI, and how to construct expressions based on these elements. It contains the following sections:

- *Permitted symbol names*
- *Program symbols* on page 1-12
- *Debugger variable symbols* on page 1-13
- *Macro symbols* on page 1-13
- *Reserved symbols* on page 1-14
- *Addresses* on page 1-22
- *Expression strings* on page 1-23
- *Target functions* on page 1-24.

1.6.1 Permitted symbol names

A symbol (also called an identifier) is a name that identifies something, for example program and debugger variables, macros, keywords, and registers.

Symbols can be up to 1024 characters in length. The first character in a symbol must be alphabetic, an underscore `_`, or the at sign `@`. The valid characters in a symbol include upper- and lower-case alphabetic characters, numeric characters, the dollar sign `$`, at sign `@`, and underscore `_`. Other symbolic characters cannot be used in symbols. The debugger distinguishes between uppercase and lowercase characters in a symbol. A symbol is therefore matched by the following regular expression:

```
[a-zA-Z_@][a-zA-Z_@$0-9]{0,1023}
```

Regular expressions are described in *Mastering Regular Expressions*.

If your compiler or assembler creates symbols that contain characters that are invalid in RealView Debugger symbols, prefix the symbol name with an `@` and enclose the rest of the name in double quotes " to reference it, for example `@"!parser"`. You cannot access a symbol including a double quote character in its name.

1.6.2 Program symbols

Program symbols are identifiers associated with a source program. They include variables, function names, and, depending on the compiler, macro names. Symbols defined in the source of the application can normally be passed to the debugger. When a program is loaded for debugging, program symbols are normally loaded into a symbol table associated with the target connection.

Some compilers insert a leading underscore _ to all program source symbols so that program symbol names are distinguished from other names. The debugger strips the first leading underscore from such program symbols when an application file is read so references to program symbols are as originally written.

Some compilers pass C and C++ preprocessor macros to the debugger. These are also usable in expressions. The debugger shows the expansion in the output.

Listing Symbols

You can list all symbols currently defined in RealView Debugger. To do this, enter:

```
printsymbols /w *
```

For more details on using this command, see `PRINTSYMBOLS` on page 2-224.

Referencing symbols

References to symbols or source-level line numbers can be unqualified or qualified. An unqualified reference includes only the symbol or line number itself. A qualified reference includes the symbol or line number preceded by a root (defined in the following section), module and/or function name. Root, module, and function names are separated from the symbol or line number by a backslash \. Module names must be in uppercase. Table 1-2 summarizes examples of qualified symbols.

Table 1-2 Qualified symbol references

Form	Example	Comment
`@`*`root`*`\\`	`@tst\\TS1ROOT`	References module `TS1ROOT` in root `@tst`. (Usually from file loaded as `tst.x` or `tst.out`.)
`\`*`global`* `::global`	`\x ::x`	References global variable `x` in current root.
`function``\`*`local`*	`main\x`	References local variable `x` in function `main`.
`MODULE``\`*`function`*	`SIEVE\main`	References function `main` in module `sieve`.
`MODULE``\`*`static`*	`SIEVE\y`	References static variable `y` in module `sieve`.

Table 1-2 Qualified symbol references (continued)

Form	Example	Comment
MODULE\line_number	`ENTRY\#18`	References line number 18 in module entry.
MODULE\function\local	`ENTRY\main\x`	References local variable x in function main in module entry.
LINE\local	`#20\x`	References local variable x in an unnamed block at line 20.

1.6.3 Debugger variable symbols

Debugger variables are created during a debugging session with the `ADD` CLI command, and all have global scope. When a debugger symbol is created you can assign it a data type (for example **`char`**, **`int`**, or **`long`**) and an initial value, but cannot assign initial values to **`struct`**, **`union`**, or **`class`** type symbols.

Debugger variables can be stored in either:

- Debugger memory. The debugger allocates memory for the variable for you.
- Target memory. You must specify a target memory address for the variable.

1.6.4 Macro symbols

A RealView Debugger macro is similar to a C function. It has a name, a return type, and optional arguments. You can also define macro-local variables, and the macro itself is a sequence of statements. Symbols are used in macros in two ways:

Macro name This identifies the macro. You are recommended to avoid using the following:

- names of the predefined macros (see Chapter 3 *RealView Debugger Predefined Macros*)
- keywords (see Chapter 4 *RealView Debugger Keywords*)
- debugger commands (see Chapter 2 *RealView Debugger Commands*)
- aliases you have defined using the `ALIAS` command (see `ALIAS` on page 2-25).

Local variables Local variables can be defined within a macro as working storage while the macro executes. A macro local variable can only be accessed by the macro in which it is defined. It is created when the macro is executed and has an undefined initial value.

All other variable macros can access and use all other symbols. Macros can call other macros, but not recursively.

See the chapter that describes working with macros in the *RealView Developer Kit v2.2 Debugger User Guide* for more information on creating and using macros.

1.6.5 Reserved symbols

Reserved symbols are reserved words that represent registers, status bits, and debugger control variables. These symbols are always recognized by the debugger and can be used at any time during a debugging session. Because reserved symbols have special meanings within the debugger command language, they cannot be defined and used for other purposes. To avoid conflict with other symbols, the names of all reserved symbols are preceded by an at sign `@`. See Table 1-3 on page 1-15 for a list of reserved symbols and their descriptions.

This section contains:

- *Displaying a list of currently defined symbols*
- *Referencing reserved symbols* on page 1-15
- *Printing reserved symbols* on page 1-16
- *Symbols for referencing processor core registers* on page 1-19
- *Symbols for referencing internal variables and board-specific registers* on page 1-20.

Displaying a list of currently defined symbols

You can display a list of the symbols that are currently defined in RealView Debugger. To do this, use the `PRINTSYMBOLS` command (see `PRINTSYMBOLS` on page 2-224):

```
printsymbols *
```

This command lists:

- RealView Debugger reserved symbols. These include symbols defined for the target associated with the current connection.
- RealView Debugger predefined macros.
- If you have an image loaded for the current connection, then the symbols defined for that image are listed.
- If you have defined any macros, then the symbols defined for those macros are listed.

Referencing reserved symbols

The RealView Debugger defines several symbols, known as reserved symbols, that retain specific information for you to access. Table 1-3 shows these reserved symbols with a short description. Reserved symbol names always begin with an at sign `@` and can be all uppercase or all lowercase.

Table 1-3 Reserved symbols

Symbol	Description
`@register`	References the named *`register`*. For more details see: • *Symbols for referencing processor core registers* on page 1-19 • *Symbols for referencing internal variables and board-specific registers* on page 1-20.

`@entry`	Used to form a function pseudo-label, *`function`*`\@entry`, that identifies the first location in the function after the code that sets up the function parameters. In general, *`function`*`\@entry` refers to either:
	• the first executable line of code in that function
	• the first auto local that is initialized in that function.
	In either case, *`function`*`\@entry` is beyond the function prologue, to ensure that the function parameters can be accessed. This enables you to set a breakpoint on a function without having to scope to the function in the File Editor pane, or without having to know an address.
	If no lines exist that set up any parameters for the function (for example, an embedded assembler function), then the following error message is displayed:
	`Error: E0039: Line number not found.`
	As an example, if you have a function `func_1(value)` you might want to set a breakpoint that triggers only when the argument `value` has a specific value on entry to the function:
	`bi,when:{value==2} func_1\@entry`

`@hlpc`	Indicates your current high-level source code line. `@hlpc` is valid only if the *Program Counter* (PC) is in a module that has high-level line information (that is, a C, C++, or assembler source module compiled with debug turned on). `@hlpc` contains the line number at the current PC only if located in source code. Otherwise, it is zero.
`@line_range`	Contains the line range of the source code associated with the PC.
`@module`	Indicates the name of the current module as determined by the location of the PC.

Table 1-3 Reserved symbols

Symbol	Description
`@procedure`	Indicates the name of the current function as determined by the location of the PC.
`@file`	Indicates the name of the current file as determined by the location of the PC.
`@root`	Indicates the current root name.

Printing reserved symbols

To print the reserved symbols, use the `FPRINTF` or `PRINTF` command with the appropriate format specifier. Alternatively, use the `PRINTVALUE` command to print the contents of a numerical reserved symbol, for example `@hlpc`. You can also use the `PRINTSYMBOLS/F` command with no arguments, and the command displays all roots. For more details on these commands, see:

- `FPRINTF` on page 2-169
- `PRINTF` on page 2-220
- `PRINTSYMBOLS` on page 2-224
- `PRINTVALUE` on page 2-229.

Also, see *Reserved symbols* on page 1-14.

Table 1-4 shows the format specifiers to use when printing reserved symbols.

Table 1-4 Format specifiers for printing reserved symbols

Information to print	Symbol to use	Format specifier
Register contents, see: • *Symbols for referencing processor core registers* on page 1-19 • *Symbols for referencing internal variables and board-specific registers* on page 1-20.	*`@register`*	This must match the type of the register.
Current instruction	`@pc`	`%m`
Current line number	`@hlpc`	`%d`
Current source line as text	`@hlpc`	`%h`
Line range of the source code identified by the PC	`@line_range`	`%s`
Current module name	`@module`	`%s`

Table 1-4 Format specifiers for printing reserved symbols

Information to print	Symbol to use	Format specifier
Current procedure name	`@procedure`	`%s`
Current file name determined by the PC	`@file`	`%s`
Current root name	`@root`	`%s`

Example

The following example shows how to use these symbols:

1. Create an include file, for example `symbols.inc`, containing the following command and macro definition (see *Macro language* on page 1-7):

   ```
   add int windowOpened=0
   define /R int rsvdSymbols(outputID)
   int outputID;
   {
     if (windowOpened = 0)
       if (outputID > 49) {
         $vopen outputID$;
         windowOpened=1;
       }
     $fprintf outputID, "root:         %s\n", @root$;
     $fprintf outputID, "hlpc:         %d\n", @hlpc$;
     $fprintf outputID, "code line:    %h\n", @hlpc$;
     $fprintf outputID, "instruction:  %m", @pc$;
     $fprintf outputID, "file:         %s\n", @file$;
     $fprintf outputID, "line_range:   %s\n", @line_range$;
     $fprintf outputID, "module:       %s\n", @module$;
     $fprintf outputID, "procedure:    %s\n", @procedure$;
     // return 0 to stop at the breakpoint
     // return 1 to continue after the breakpoint
     return (0);
   }
   .
   ```

2. Connect to RVI-ME (see `CONNECT` on page 2-98)

3. Load the Dhrystone image from the main examples directory (see `LOAD` on page 2-189):

   ```
   load/r 'main_examples_directory\dhrystone\Debug\dhrystone.axf'
   ```

4. Include the file you created in step 1 to define the macro (see `INCLUDE` on page 2-181). If this is in the directory `c:\myscripts`, then enter:

```
include 'c:\myscripts\symbols.inc'
```

5. Run the macro, with an `outputID` of 20, the Standard I/O window (see *Window and file numbers* on page 1-5):

   ```
   macro rsvdSymbols(20)
   ```

   The following details are displayed:

   ```
   root:        @dhrystone
   hlpc:         0
   code line:    <invalid line>
   instruction: $00008000 EAFFFFFF  B        $L0x8004 $      ~<S0x8004>
   file:        ../../angel/startup.s
   line_range:
   module:      STARTUP_S
   procedure:   __main
   ```

   Here the high-level source code line is zero, and no code line can be displayed. This is because the PC is at a location in a library module that was compiled without debug turned on, and the source file is not available.

6. Set a breakpoint at the first executable line of code in function main, that also runs the `rsvdSymbols()` macro when the breakpoint is reached (see `BREAKINSTRUCTION` on page 2-61):

   ```
   breakinstruction,macro:{rsvdSymbols(20)} main\@entry
   ```

7. Run the Dhrystone application (see `GO` on page 2-173):

   ```
   go
   ```

   When the breakpoint is reached, the `rsvdSymbols()` macro runs, and the following details are displayed:

   ```
   root:        @dhrystone
   hlpc:        91
   code line:      Next_Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
   instruction: 000084D0 E3A00030  MOV      r0,#0x30
   file:        c:\program files\arm\rvd\examples\1.7\5\windows\dhrystone\
   dhry_1.c
   line_range: 91..91
   module:      DHRY_1
   procedure:   main
   ```

8. Reload the Dhrystone application and clear the previous breakpoint (see `RELOAD` on page 2-239 and `CLEARBREAK` on page 2-94):

   ```
   reload
   clear
   ```

9. Set the breakpoint again, but specify the module and line:

   ```
   breakinstruction,macro:{rsvdSymbols(20)} \DHRY_1\#91:1
   ```

10. Run the Dhrystone application:

    ```
    go
    ```

    The listing shown in step 7 is displayed, but the `line_range` is now shown as `79..91`.

Symbols for referencing processor core registers

Table 1-5 shows the symbols to use if you want to reference the processor core registers. These are the registers shown in the **Core** tab of the Register pane. You can also perform operations on these registers, see *Operations on symbols and registers* on page 1-21.

Table 1-5 Processor core register symbols

Register symbol	Description
`@R`*n*	Use this symbol to reference registers `r1` to `r12`.
`@R13` or `@SP`	References the `SP` register.
`@R14` or `@LR`	References the `LR` register.
`@R15` or `@PC`	References the `PC` register.
`@CPSR`	References the `CPSR` register.
`@CPSR_FLG`	References the NZCV flags of the CPSR.
`@CPSR_F`	References the FIQ register of the CPSR.
`@CPSR_I`	References the IRQ register of the CPSR.
`@CPSR_T`	References the STATE register of the CPSR.
`@CPSR_MODE`	References the MODE register of the CPSR.
`@R8_`*bank* `@R9_`*bank* `@R10_`*bank* `@R11_`*bank* `@R12_`*bank*	References register `R8`, `R9`, `R10`, `R11`, and `R12` in register banks *bank*. These are used only in register banks `USR` and `FIQ`. For example, `@R9_USR`.
`@R13_`*bank*	References the `SP` register in register bank *bank*. For example, `@R13_IRQ`.
`@R14_`*bank*	References the `LR` register in register bank *bank*. For example, `@R14_SVC`.

Table 1-5 Processor core register symbols (continued)

Register symbol	Description

`@SPSR_`*`bank`*
`@SPSR_`*`bank_regs`*
	References the SPSR registers in a register *`bank`*. *`regs`* is one of the following:
	`FLG` `NZCV` flags
	`F` `FIQ` register
	`I` `IRQ` register
	`T` `STATE` register
	`MODE` `MODE` register.
	Note
	There is no `SPSR` register in the `USR` bank.
	For example, `@SPSR_IRQ_T` references the processor `STATE` register in the `IRQ` register bank.

Symbols for referencing internal variables and board-specific registers

You can also reference internal debugger variables and board-specific registers:

- Internal debugger variables are displayed in extra tabs of the Register pane, and depend on your target connection. For example, the **Debug** tab is available when connecting to RealView ICE.
- Board-specific registers are displayed in other tabs of the Register pane.

Note

You can also perform operations on the internal debugger variables and board-specific registers, see *Operations on symbols and registers* on page 1-21.

To find the symbol names for the internal debugger variables, you must use the RealView Debugger GUI. To find the symbol names for board-specific registers, you can use either the RealView Debugger GUI, or look in the related board/chip definition file (`.bcd`) in the RealView Debugger *`program_directory`*`\etc` directory.

To find the name of a board-specific register or internal debugger variable using the RealView Debugger GUI:

1. Select the required tab, for example, **Debug**.
2. Right-click on the register or variable that you want to reference, for example, `semihost_enabled`.

3. Select **Properties** from the context menu.

   This displays an Information dialog. The Register: field shows the symbol name. For `semihost_enabled`, the symbol name is `@SEMIHOST_ENABLED`.

To find the name of a board-specific register from the related board/chip definition file:

1. Find the file in the RealView Debugger *`program_directory`*`\etc` directory that has the same name as the board/chip definition. For example, the names for the Integrator AP board are defined in the file `AP.bcd`.

2. Open the file with a text editor.

   Note

   Do not make changes to this file directly. Use the Connection Properties window in the GUI to make any changes. See the chapter that describes configuring custom targets in *RealView Developer Kit v2.2 Debugger User Guide*.

3. Search for the line containing `Register_Window.`*`boardname`*, where *`boardname`* is the name of the board, for example `Register_Window.AP`. The lines following this entry contain the register names.

1.6.6 Operations on symbols and registers

You can perform operations on symbols, on the registers listed in Table 1-5 on page 1-19, the internal debugger variables, and board-specific registers. The operations you can perform are listed in Table 1-6.

Table 1-6 Register operations

Operation	Description	Examples
`@var = value`	Assign a value to the symbol.	`@PC = 0x8000`
`@var++`, `@var--`	Increment or decrement the value in the symbol.	`@R6++`

`@var = @var + value`
`@var = @var - value`
	Add a value to, or subtract a value from, the symbol.	`@R12 = @R11+2`

`@var = @var * value`
`@var = @var / value`
	Multiply or divide the value in the symbol by a specified `value`. Dividing by zero gives an error message.	`@R7 = @R7*2`

Table 1-6 Register operations (continued)

Operation	Description	Examples
`@var &= [~]mask`	AND the `mask` value with the contents of the symbol. ~ indicates the inverse of the mask value.	`@FLG &= 3` `@FLG &= ~3`
`@var |= [~]mask`	OR the `mask` value with the contents of the symbol. ~ indicates the inverse of the mask value.	`@FLG |= 3`
`@var ^= [~]mask`	Exclusive OR the `mask` value with the contents of the symbol. ~ indicates the inverse of the mask value.	`@FLG ^= 3`

1.6.7 Addresses

An address can be represented by most C expressions that evaluate to a single value. In source-level mode, expressions that evaluate to a code address cannot contain numeric constants or operators, unless you use a cast.

Data address and assembly-level code address expressions can also be represented by most legal C expressions. For details on legal C expressions, see the *C language Reference Manual*. There are no restrictions involving constants or operators. Table 1-7 summarizes the special addressing types supported by the RealView Debugger.

Table 1-7 Address expressions

Addressing type	Indicator	Example
Indirect addresses	`[ ]`	`PRINTVALUE (H W) [23]`
Line numbers	`#`	`BREAKINSTRUCTION #10`
Address ranges	`..`	`DUMP 0x2200..0x2214 DUMP 0x2200..+14`
Multi-statement reference	`:`	`BREAKINSTRUCTION #21:32` (refers to the statement on line 21 that contains column 32)
	`.`	`BREAKINSTRUCTION #21.2` (refers to the second statement on line 21)
Address of non-label symbol. The symbol cannot be that of a register or a constant.	`&`	`BREAKREAD &var`

1.6.8 Expression strings

An expression string is a list of values separated by commas. The expression string can contain expressions and ASCII character strings enclosed in quotation marks. For several commands, each value in an expression string can be changed to the size specified by the size qualifiers. If the size is changed, padding is added to elements that do not fit.

Examples of expression strings are shown in Table 1-8.

Table 1-8 Examples of expression strings

String	Results
`1,2,"abc"`	Values 1 and 2, and ASCII values of abc.
`3+4, count, foo()`	Value 7, value of count, results of calling foo.
`'1xyz123'`	ASCII values of 1, x, y, z, 1, 2, and 3.

You can cast values to arrays, so that for example you can access the second byte of a 32 bit word by casting the word to a byte array.

Note

If you enter a command line that starts with an open-bracket `(`, or an asterisk `*`, the debugger interprets this as if you had entered a `CEXPRESSION` command with that text as its argument (see `CEXPRESSION` on page 2-91). For example:

```
> *(char*)0x8000 = 0
```

is equivalent to:

```
> CE *(char*)0x8000 = 0
```

As with the normal `CEXPRESSION` command, you can use this to view or modify program variables and memory. `CE` is the abbreviation for `CEXPRESSION`.

1.6.9 Target functions

Target functions can be called within a debugger expression. Depending on the processor, the data type of the return value determines the type that the function call takes. Public labels, like runtime routines, are assumed to return an integer. Passing structures by returning a pointer is not supported.

A target function called from a debugger expression behaves the same way as if it were called from a user program. Target functions can invoke I/O operations or error messages in addition to activate breakpoints and associated macros. If a breakpoint is hit that stops execution, the registers are restored and the call ends with an error.

Arguments are copied to the stack below the current function. The return address is the entry point of the application where a breakpoint is placed.

Macros take higher precedence than target functions. If a target function and a macro have the same name, the macro is executed unless the target function is qualified. For example, `strcpy` is a predefined debugger macro, while `PROG\strcpy` is a function within the module `PROG`. The predefined macro is referenced as `strcpy(t,s)`, while `PROG\strcpy(t,s)` refers to the function within `PROG`. A target function must be called within a debugger expression that is used within a command. It cannot be directly executed as a command, but a macro can.

Example 1-1 Calling a target function

```
CE PROG\strcpy(t,s)
```

1.7 Using variables in the debugger

It is important to understand how to access variables that are stored in memory. This section describes symbol storage classes and data types. It describes how to qualify a symbolic reference with a module or function name, how to specify fully referenced variables, and how to make stack references. It contains the following sections:

- *Scope*
- *Data types* on page 1-26
- *Root names* on page 1-28
- *Module names* on page 1-29
- *Variable references* on page 1-29
- *Stack references* on page 1-31.

1.7.1 Scope

All variables and functions in a C or C++ source program have a storage class that defines how the variable or function is created and accessed. C preprocessor symbols might not be available to the debugger.

Global (`extern`)

In the debugger, global variables can be referred to from any module. However, if a symbol of the same name exists in the local scope, this variable must be qualified by a root name, by \ (current root), or with `::`.

Static In the debugger, static functions can be referred to from the same module without qualification. Static functions in other modules must be qualified with the module name if the name is ambiguous or the module has not been used yet (not loaded).

Local A local variable is accessible when it is local to the current function, local to the current unnamed block, or when its function is on the stack. It can be qualified by function, line, or stack level.

Register Register variables might not be available from all lines in the function, because hardware registers can be shared by more than one local register variable. A register variable is accessible when it is local to the current function or when its function is on the stack. It can be qualified by function or stack level.

Scoping rules

References to symbols follow the standard scoping rules of C and C++. If a symbol is referenced, the debugger searches its symbol table using the following priority:

1. Any symbol local to the current macro.
2. Any symbol local to the current line.
3. Any symbol local to the current function.
4. Any symbol local to the class of the current function.
5. Any symbol static to the current module.
6. Any global symbol not necessarily in the current module.
7. A static symbol in another module.
8. A global symbol in another root (that is, a different loaded file).

1.7.2 Data types

All symbols and expressions have an associated data type. Source language modules can contain any valid C or C++ language data type. Assembly language modules can contain variables with the types byte, word, long, 8-byte long, single-precision floating point, double-precision floating point, or label. Some assemblers might have other types such as fixed-point. These types are treated by the debugger as **`unsigned char`**, **`unsigned short int`**, **`unsigned long`**, **`long long`**, **`float`**, **`double`**, and **`label`**, respectively. In addition, each symbol has an attribute that indicates whether a variable was defined in a code or data area. Also, the assembler can create arrays of these types in addition to structures (check with the assembler manufacturer for details).

Type conversion

The RealView Debugger performs data-type conversions under the following circumstances:

- when two or more operands of different types appear in an expression, data type conversion is performed according to the rules of C or C++
- when arguments are passed to a macro or target function, the types of the passed arguments are converted to the types given in the macro function definition
- when the data type of an operand is forced by user-specified type casting, it is converted

- when a specific type is required by a command, the value is converted according to the rules of C/C++.

Type casting

Type casting forces the conversion of an expression to the specified data type. The contents of any variables that are referenced are not altered. Debugger expressions can be cast into different types using the following syntax:

(*type_name*) *expression*

Example 1-2 Casting symbols and expressions into different types

```
(char) prime            /* prime is cast to type char */
(float) 12              /* value is 12.0. (integer 12 in floating point) */
(int) sin(0.2)          /* value is 0, sin(0.2) is 0.198, truncates to 0 */
(int) ptr_char          /* the variable expression ptr_char is */
                        /* cast to type int */
```

The debugger can cast some expression types to an array type. Example 1-3 casts the constant expression 7 to an array of three characters starting at location `0x0007`.

Example 1-3 Casting to an array

```
(char[3]) 7             /* address is 0x0007 */
```

This type of casting to an array can be used with the `PRINTVALUE` command (see page 2-229). Assembly language structures can be displayed in a more meaningful form by using this technique. Table 1-9 lists additional special casting types. Arrays of hexadecimal types and pointers to hexadecimal types can also be used.

Table 1-9 Special casting types

Cast	Commands	Meaning
`(QUOTED STRING)` or `(Q S)`	`PRINTVALUE`	Show as "string."
`(INSTRUCTION ADDRESS)` or `(I A)`	All	Convert into a legal source-level address.
`(UNKNOWN TYPE)` or `(U T)`	All	Convert into a single byte.

Table 1-9 Special casting types (continued)

Cast	Commands	Meaning
`(HEX BYTE)` or `(H B)`	All	Show in hex bytes.
`(HEX WORD)` or `(H W)`	All	Show in 16 bit hex.
`(HEX DOUBLE WORD)` or `(H D)`	All	Show in 32 bit hex.

1.7.3 Root names

Root names indicate the top level in a qualified path name. Each time the debugger is invoked, it automatically creates a base root. This root is assigned the name \\ and contains all debugger variables, macros, and most user-defined symbols. The only user-defined symbols that are not in the base root are those created with the `ADD` command. The remainder are built-in (see `ADD` on page 2-18).

When an executable program is loaded, the debugger automatically creates a second root for that program. The name of this root is the name of the program with an at sign `@` prepended to it. For example, when the debugger loads the `proga` program, it creates the root `@PROGA`. An alternative root name can be specified with the `LOAD` command (see `LOAD` on page 2-189).

If two programs have the same name, the debugger appends an underscore followed by a number (that is, `@`*`NAME_1`*, `@`*`NAME_2`*) to the second (and any subsequent) program.

To specify which root a module belongs to, use `@`*`ROOT`*`\\`*`MODULE`* where *`ROOT`* is the root name and *`MODULE`* is the module name. The \\ specifies that the preceding symbol is a root name. Use \\ to specify the base root, which contains built-in type, macro, and reserved word information. In the `PRINTSYMBOLS` command, the root can be specified directly. The reserved symbol `@ROOT` points to the current root name. For more information about debugger reserved symbols, see *Reserved symbols* on page 1-14.

Example 1-4 Using root names

```
ps \                        /* Shows all symbols in current root */
ps/t \\                     /* Shows types in base root */
ps/m @sieve\\               /* Shows all modules in root @sieve */
ps/f                        /* Shows all roots */
```

The debugger considers the context to help determine the current root. If the context is within a module, the root of that module is the current context. The use of a backslash \ refers to the current root, as specified by the context.

1.7.4 Module names

Module names qualify symbolic references. The module name is usually the source filename without the extension. If the extension is not standard (that is, `.c` for C language programs), the extension is preserved, and the dot . is replaced with an underscore _. For example:

- `SIEVE\main`
- `SIEVE_H\#4`
- `PORT_ASG\x`

This convention avoids a conflict with the C period operator ., that indicates a structure reference. Module names are changed as follows:

- `SIEVE.C` becomes `SIEVE`
- `SIEVE.H` becomes `SIEVE_H`

All module names are converted to uppercase by the debugger. To avoid confusion, it is recommended that function names are not all uppercase. If two or more modules have the same name, the debugger appends an underscore followed by a number to the second, and any subsequent, module (for example, `PROGA_1`, `PROGA_2`, and `PROGA_3`). To see the current module and function, use the `CONTEXT` command.

See *Printing reserved symbols* on page 1-16 for details on how to print the current module and functions names.

1.7.5 Variable references

In C, using a variable in an expression can result in a value or an address. A fully referenced variable results in a value. A partially referenced variable results in an address. Some legal assembly language variables can conflict with C operators, such as dot `.` and question mark ?. These characters are replaced with an underscore `_`. Examples of variable references are provided in Table 1-10, including an indication of what type of reference is being made.

Table 1-10 Examples of references to variables

Variable reference	Reference type

`int A;`
`A = 5;`
	`A` is fully referenced.

`long temp;`
`temp = 9;`
	`temp` is fully referenced.

`int arr[10], *LABEL;`	`arr` is not fully referenced so its address is used.

Table 1-10 Examples of references to variables

Variable reference	Reference type

`LABEL = arr;`
`arr[3] = 8;`
	`arr[3]` is fully referenced.

`int AB[10][10], *LABX;`	`AB` is not fully referenced so its address is used.

`LABX = AB[5];`
`LABX = LABEL;`
	`LABEL` is fully referenced so its value is used (the address it points to).

`char *p,c;`
`p = &c;`
	`p` is fully referenced. `c` is not fully referenced.

`c = *LABEL;`	`LABEL` is dereferenced so the value of its address is used.

When you refer to a variable in a C/C++ expression that is not fully referenced, you are actually referring to the address of that variable, not the value. For this reason, the variable is considered unreferenced. The normal C operators are implemented to modify references, as shown in Table 1-11.

Table 1-11 C operators for referencing and dereferencing variables

Operator	Scalar	Pointer	Array	Structure	Union
`*`	-	Ref	Ref	-	-
`&`	Deref	Deref	-	Deref	Deref
`->`	-	Ref[a]	-	-	-
`.`	-	-	-	Ref	Ref
`[ ]`	-	Ref	Ref	-	-

a. Must be a pointer to a structure or union. The right side must be a member of that structure or union. Otherwise, it is illegal.

These operators let you reference, or get the value of, and dereference, or get the address of, variables. The concept of referenced and dereferenced variables also applies to breakpoints. For example:

```
BA arrayname
```

This command sets an access breakpoint at the address of the array `arrayname` because `arrayname` is not fully referenced. By including the special operator `&`, the following command enables you to set a breakpoint at the value of `arrayname[3]`:

```
BA &arrayname[3]
```

The following form of the command sets a breakpoint at the value stored in `arrayname[3]` and not the address of `arrayname[3]`, because it is now fully referenced.

```
BA arrayname[3]
```

1.7.6 Stack references

When a function is invoked in C/C++, space is allocated on the stack for most local variables. Typically, space is also allocated for a return address for returning to the calling routine. If a function calls another function, all information is saved on the stack to continue execution when the called function returns. The function is now nested.

You can reference variables and functions nested on the stack implicitly or explicitly.

Implicit stack references

Within the debugger, you can implicitly reference variables on the stack as follows:

- To refer to variables on the stack in the current function, specify the name of the variable, for example `x`.
- To refer to a local variable in a nested function, specify the function name followed by a backslash and the name of the local variable (`main\i` for example). If the nested function is recursive, the last occurrence of that function is used. An explicit reference enables any occurrence to be selected.

Explicit stack references

A function is allocated storage for its variables on the stack when it is currently executing. To refer to variables on the stack explicitly, you must specify the nesting level of the function preceded by an at sign `@`. The Call Stack window in source-level mode displays nesting level information. The current function is `@0`, its caller is `@1`.

You can reference functions on the stack as follows:

- To refer to the address where some function on the stack returns, specify the function nesting level preceded by an at sign @. For example, `GO @1` executes the program until the debugger reaches the address that corresponds to the location where the current function returns to its caller (the instruction after the call). The `LIST` and `DISASSEMBLE` commands can be used to show the code at the return address (`LI @2` for example).

  In nonrecursive programs, the command `GO @1` corresponds to setting a breakpoint when the current function returns to its caller. In recursive programs, the address alone might not be enough to specify the instance that you want. A command such as `GO@1; until(depth == 4)` can be used to specify the instance of the address that you want (assuming `depth` is a local variable in your recursive function that determines the instance you are executing).

- To explicitly refer to a local variable in a nested function, specify the function nesting level followed by a backslash and the name of the variable. For example, `PRINTVALUE @3\str` references the local variable `str` of the function at nesting level 3.
- To see all available information about a function, specify the `EXPAND` command followed by the function nesting level. For example, `EXPAND @7` displays all information about the function at the specified level for that particular invocation. This information includes the name of the function, the address that is returned to, and all local variables in the function and their values.

1.8 Source patching with macros

When debugging your application program, sometimes errors can be temporarily patched with source statements. It is often unnecessary to edit the source code, and recompile and link. Instead, you can use a temporary patch by using macros with breakpoints.

To insert a few lines of source code in your program:

1. Define a macro containing the statements that you want to insert.
2. Start a debugging session and set a breakpoint on the source line following the point where you want to insert the new lines.
3. Attach your macro to this breakpoint. See Chapter 2 *RealView Debugger Commands* for details explaining how to do this using the breakpoint commands. Alternatively, see the *RealView Developer Kit v2.2 Debugger User Guide* for details explaining how to do this in the GUI.
4. Run the program until execution stops at the breakpoint.
5. The source statements in your macro are interpreted and executed. The macro completes.
6. Program execution continues normally.

Note

Using a macro in this way might cause problems with compiler optimizations, for example the ordering of instructions might have been altered by the compiler.

You can also use a similar approach to jump over or skip lines of source code:

1. Define a macro to set the PC to a point beyond the lines that are not executed.
2. Start a debugging session and set a breakpoint on the first line to be skipped.
3. Attach your macro to this breakpoint.
4. Run the program until execution stops at the breakpoint.
5. The source statements in your macro are interpreted and executed. The macro completes.
6. Program execution continues normally from the new position of the PC.

You can also use the JUMP command for looping and skipping over commands, shown in the fragment in Example 1-5. The JUMP command takes a label and an expression. If the expression evaluates to True then control jumps to the specified label. If the label is positioned earlier in the file, this loops. If the label is positioned later in the file, all intermediate commands are skipped.

The expression can test:

- symbols, using the isalive() keyword (see **isalive** on page 4-12)
- results
- local symbols, created with ADD
- file tests, using macros.

Example 1-5 Using the JUMP command

```
add int cnt = 20
initialize
:repeat                           /* loop 20 times */some_commands
jump repeat,cnt                   /* repeat until cnt==0 */
;
; define some local vars if not defined.
;
jump nodefine,isalive(cnt)==1
some_commands
:nodefine
```

For more information on the ADD and JUMP commands, see ADD on page 2-18 and JUMP on page 2-186.

1.8.1 Patching example to reset Program Counter

The source code being debugged contains the following lines:

```
24
25 count = 5;
26 for (i=0; i < MAXNUM; i++)
27 {
28     array[i]=1;
29     count=count+2;
30     k=count*i;
31 }
32
```

To jump over or skip lines 29 and 30, and to insert a new line temporarily incrementing `count` by 1:

1. Define a macro that contains statements to increment `count` and move the PC over the two lines:

   ```
   DEFINE patch_29()
   {
       count++;            /* increment count by 1 */
       $SETREG @PC = #31$; /* reset program counter so skipping 29 & 30 */
       return(1);          /* return 1 to continue normal execution */
   }
   .
   ```

2. Start a debugging session and set an instruction breakpoint on line 29. See *Execution control* on page 2-4 for a list of the commands you can use to set breakpoints. Alternatively, see the *RealView Developer Kit v2.2 Debugger User Guide* for details on how to set breakpoints using the GUI interface.

3. Attach your macro to this breakpoint.

4. Run the program until execution stops at the breakpoint.

5. The source statements in your macro are interpreted and executed. The macro completes.

6. Program execution continues normally.

1.8.2 Patching example to emulate a serial port

To emulate a serial port in your source code:

1. Define a macro that emulates a serial port:

```
add unsigned long last_time;       /* create local symbol */
define int ser_port(offset,base)   /* macro definition */
    int offset;                /* offset of device register */
    unsigned short *base;      /* base of port */
{
    unsigned short value;
    if (offset == 0)
    {                           /* control register */
        if (last_time && @cycle-last_time < 20)
        {
            error (0, “ser_port: access less than
                allowed time: %d”, @cycle-last_time);
            return (0);
        }
        last_time = @cycle;
        value = base[offset];  /* cache written value */
        base[offset] = 0;      /* reset */
        if (value == 0x20)
        {                      /* want to read */
            ...
        }
        ...
    }
    ...
}
.
```

2. Start a debugging session and set a breakpoint on the source code to stop execution immediately before it accesses the serial port, for example at line 20, and attach your macro to this breakpoint:

```
bi \module_name\#20 ;ser_port(0,&ser_port)
```

3. Continue debugging using the newly-inserted serial port emulation.

As with the previous example, this is only a temporary patch so the source code must be edited and then recompiled. Be careful, however, when using such a patch in optimized code.

1.8.3 Other ways to use macros

This section describes other ways that you can use macros. It includes:

- *Using macros to interact with files and windows*
- *Using macros with commands*
- *Sending debug information to the Output pane* on page 1-38
- *Interacting with a user* on page 1-38.

Using macros to interact with files and windows

During your debugging session, you can use macros to read from or write to a file, or to write to a window. The macros to do this are listed in Table 1-12.

Table 1-12 Macros for interacting with files and windows

Macro	Acts on	See
`fopen()`	Files	`fopen` on page 3-15
`fgetc()`	Files	`fgetc` on page 3-12
`fputc()`	Files	`fputc` on page 3-17
`fread()`	Files	`fread` on page 3-19
`fwrite()`	Files and windows	`fwrite` on page 3-21

Using macros with commands

You can use macros with the commands listed in Table 1-13

Table 1-13 Macro related commands

Command	See
`BGLOBAL`	`BGLOBAL` on page 2-38
`BREAKACCESS`	`BREAKACCESS` on page 2-43
`BREAKEXECUTION`	`BREAKEXECUTION` on page 2-52
`BREAKINSTRUCTION`	`BREAKINSTRUCTION` on page 2-61
`BREAKREAD`	`BREAKREAD` on page 2-69
`BREAKWRITE`	`BREAKWRITE` on page 2-78

Table 1-13 Macro related commands (continued)

Command	See
`DELETE`	`DELETE` on page 2-114
`GO`	`GO` on page 2-173
`GOSTEP`	`GOSTEP` on page 2-175
`MACRO`	`MACRO` on page 2-195
`SHOW`	`SHOW` on page 2-269
`VMACRO`	`VMACRO` on page 2-341

Sending debug information to the Output pane

You can send debugging information to the Output pane in the GUI with the `error()` predefined macro (see `error` on page 3-10).

Interacting with a user

A number of macros are provided that enable a user to interact with your script and then continue execution based on the decision, or data, entered. See the following sections:

- `prompt_file` on page 3-32
- `prompt_list` on page 3-34
- `prompt_text` on page 3-36
- `prompt_yesno` on page 3-38
- `prompt_yesno_cancel` on page 3-40.

You can also use these predefined macros with other macros and include files. If using these predefined macros with the `MACRO` command (see `MACRO` on page 2-195) in include files, use the `JUMP` command (see `JUMP` on page 2-186) to implement the user's decision. The following example shows how you can use these predefined macros with the `MACRO` command:

Example 1-6 Using the prompt macros with JUMP

```
add int val
add char buff[15]
strcpy(buff, “one\ntwo\nthree”)

// Implement user’s choice
define /R void choice(option)
```

```
int option;
{
  if (choice > 0) {
    if (choice == 1)
      $printf "Item one selected.\n"$;
    else if (choice == 2)
      $printf "Item two selected.\n"$;
    else
      $printf "Item three selected.\n"$;
  }
}
.
// Choose an option
:repeat
  macro val = prompt_list("Choose one:", buff)
  choice(val)
  jump repeat, val>0                 // Repeat until user clicks Cancel
```

Chapter 2
RealView Debugger Commands

This chapter describes available RealView® Debugger commands. It contains the following sections:

2.1 Command syntax definition

Many commands have alternative names, or aliases, that you might find easier to remember. Command names and aliases can be abbreviated. For example, `ADDBOARD` can be abbreviated to `ADDBO`. The syntax definition for each command shows how it can be shortened by underlining the abbreviation, that is <u>`ADDBO`</u>`ARD`.

In the syntax definition of each command:

- square brackets `[...]` enclose optional parameters
- words enclosed in braces `{}` separated by a vertical bar `|` indicate alternatives from which you choose one
- parameters that can be repeated are followed by an ellipsis `...`.

Do not type square brackets, braces, or the vertical bar. Replace parameters in *`italics`* with the value you want. When you supply more than one parameter, use a comma or a space or a semicolon as a separator, as shown in the syntax definition for the command. If a parameter is a name that includes spaces, enclose it in double quotation marks.

2.1.1 Specifying address ranges

Many commands enable you to specify a range of addresses. You specify an address range using either of the following formats:

`start_addr..end_addr`

Start address and an absolute end address, for example:

```
memmap,define 0x10000..0x20000
```

`start_addr..+length`

Start address and length of the address region, for example:

```
memmap,define 0x10000..+0x10000
```

In both cases, the start and end values are inclusive.

You can also use symbol names, such as function names, as the start address. For example:

```
printdsm main..+1000
```

```
fill Arr_2_Glob..+64=0xFF
```

2.2 Debugger commands listed by function

This section lists the commands according to their general function:

- *Board file access*
- *Execution control* on page 2-4
- *Examining source files* on page 2-5
- *Program image management* on page 2-6
- *Target registers and memory* on page 2-7
- *Status enquiries* on page 2-8
- *Macros and aliases* on page 2-8
- *CLI* on page 2-9
- *Program symbol manipulation* on page 2-9
- *Creating and writing to files and windows* on page 2-10
- *Processor tracing* on page 2-11
- *RTOS debugging* on page 2-11
- *Miscellaneous* on page 2-12.

This section does not include command aliases. See *Alphabetical command reference* on page 2-13 for a full, alphabetical list of commands.

2.2.1 Board file access

Table 2-1 shows the commands that operate on boards, that is target processors, development systems and their subcomponents.

Table 2-1 Board file access commands

Description	Command
Add target definition	`ADDBOARD` on page 2-21
Connect the debugger to a target	`CONNECT` on page 2-98
Remove target definition	`DELBOARD` on page 2-113
Disconnect the debugger from a target	`DISCONNECT` on page 2-124
List board descriptions	`DTBOARD` on page 2-133
Write board memory map as linker file	`DUMPMAP` on page 2-143
Edit current board definition	`EDITBOARDFILE` on page 2-146
Read, or reread, a board file	`READBOARDFILE` on page 2-234

2.2.2 Execution control

Table 2-2 shows the commands that control target execution, including instruction and data breakpoints.

Table 2-2 Execution control commands

Description	Command
Initialize or reset the processor	`EMURESET` on page 2-148 `RELOAD` on page 2-239 `RESET` on page 2-241 `RESTART` on page 2-245
Start executing from current state	`GO` on page 2-173 `RUN` on page 2-250
Set a data or instruction breakpoint	`BREAKACCESS` on page 2-43 `BREAKEXECUTION` on page 2-52 `BREAKINSTRUCTION` on page 2-61 `BREAKREAD` on page 2-69 `BREAKWRITE` on page 2-78
Set a data or instruction tracepoint	`TRACE` on page 2-288 `TRACEDATAACCESS` on page 2-301 `TRACEDATAREAD` on page 2-307 `TRACEDATAWRITE` on page 2-313 `TRACEEXTCOND` on page 2-318 `TRACEINSTREXEC` on page 2-323 `TRACEINSTRFETCH` on page 2-328
Clear, enable or disable a breakpoint	`CLEARBREAK` on page 2-94 `DISABLEBREAK` on page 2-120 `ENABLEBREAK` on page 2-150 `RESETBREAKS` on page 2-243
Display currently set breakpoints	`DTBREAK` on page 2-135
Stop execution at current point	`HALT` on page 2-177 `STOP` on page 2-283
Set or change processor exceptions	`BGLOBAL` on page 2-38
Step by instruction	`STEPINSTR` on page 2-273 `STEPOINSTR` on page 2-278

Table 2-2 Execution control commands (continued)

Description	Command
Step by source line	STEPLINE on page 2-275 STEPO on page 2-280
Step invoking a macro at each step	GOSTEP on page 2-175
Do something when execution starts or stops	ONSTATE on page 2-209

2.2.3 Examining source files

Table 2-3 shows the commands that let you examine the program source files.

Table 2-3 Examining source file commands

Description	Command
Display a specific source file	LIST on page 2-187
Display execution context	CONTEXT on page 2-102 DOWN on page 2-131 UP on page 2-335 WHERE on page 2-348
Display locals of an execution context	EXPAND on page 2-157
Select source or assembly display	MODE on page 2-203
Move the display location within program	SCOPE on page 2-252
Display program source files	DTFILE on page 2-137

2.2.4 Program image management

Table 2-4 shows the commands that manipulate image (executable) files.

Table 2-4 Program image management commands

Description	Command
Reload or restart current executable	`RELOAD` on page 2-239 `RESTART` on page 2-245
Add or remove executable file from loaded files list	`ADDFILE` on page 2-23 `DELFILE` on page 2-116
Load target with one or more executable files	`LOAD` on page 2-189 `RELOAD` on page 2-239
Unload an executable file or process	`UNLOAD` on page 2-333
Write to FLASH memory	`FLASH` on page 2-164
Translate host filesystem pathnames	`NAMETRANSLATE` on page 2-206 `PATHTRANSLATE` on page 2-215
Define program (argc, argv)	`ARGUMENTS` on page 2-34
Define run mode	`RUN` on page 2-250
Disassemble target memory	`DISASSEMBLE` on page 2-122
Print disassembled target memory	`PRINTDSM` on page 2-218
Verify data or image file against memory	`VERIFYFILE` on page 2-338
Display more information about load errors	`DLOADERR` on page 2-127
Do something when execution starts or stops	`ONSTATE` on page 2-209

2.2.5 Target registers and memory

Table 2-5 shows the commands that manipulate target registers and memory.

Table 2-5 Target register and memory access commands

Description	Command
Enable and change target memory layout	`MEMMAP` on page 2-197
Fill target memory with value or values	`FILL` on page 2-161 `SETMEM` on page 2-257 `CEXPRESSION` on page 2-91
Copy or compare target memory areas	`COPY` on page 2-104 `COMPARE` on page 2-96
Change target registers	`SETREG` on page 2-259
Display memory in window	`MEMWINDOW` on page 2-201 `LIST` on page 2-187
Disassemble target memory	`DISASSEMBLE` on page 2-122
Search for value or values in memory	`SEARCH` on page 2-254
Write to FLASH memory	`FLASH` on page 2-164
Write memory map to linker file	`DUMPMAP` on page 2-143
Write host file into target memory	`READFILE` on page 2-235
Write target memory to screen	`DUMP` on page 2-141
Write target memory to host file	`WRITEFILE` on page 2-351
Compare host file with target memory	`TEST` on page 2-285 `VERIFYFILE` on page 2-338

2.2.6 Status enquiries

Table 2-6 shows the commands that display information about the current debugger session.

Table 2-6 Status enquiry commands

Description	Command
Display current image file information	`DTFILE` on page 2-137
Display more information about load errors	`DLOADERR` on page 2-127
Display execution context	`CONTEXT` on page 2-102 `WHERE` on page 2-348
Display currently set breakpoints	`DTBREAK` on page 2-135
Display trace status	`DTRACE` on page 2-139
Display board descriptions	`DTBOARD` on page 2-133
Display the contents of a macro	`SHOW` on page 2-269
Display and define user preferences	`OPTION` on page 2-211 `SETTINGS` on page 2-263

2.2.7 Macros and aliases

Table 2-7 shows the commands that define and display command aliases and macros. More information about macros can be found in Chapter 1 *Working with the CLI* and also in the *RealView Developer Kit v2.2 Debugger User Guide*.

Table 2-7 Macro and alias commands

Description	Command
Define a command macro	`DEFINE` on page 2-110
Invoke a command macro	`MACRO` on page 2-195
Step invoking a macro at each step	`GOSTEP` on page 2-175
Define, list, delete command alias	`ALIAS` on page 2-25
Attach macro to window with auto-update	`VMACRO` on page 2-341
Display the contents of a macro	`SHOW` on page 2-269

2.2.8 CLI

Table 2-8 shows the functions that manipulate the command line itself.

Table 2-8 CLI commands

Description	Command
Run script file	`INCLUDE` on page 2-181
Define error action for script file	`ERROR` on page 2-152
Cause an abnormal error for script file	`FAILINC` on page 2-160
Interrupt current asynchronous command	`CANCEL` on page 2-89 `INTRPT` on page 2-183
Jump (goto) another point in the script	`JUMP` on page 2-186
Log CLI actions to file	`JOURNAL` on page 2-184 `LOG` on page 2-193
Log STDIO messages to a file	`STDIOLOG` on page 2-271

2.2.9 Program symbol manipulation

Table 2-9 shows the commands that display and change symbols the debugger symbol table.

Table 2-9 Program symbol manipulation commands

Description	Command
Create symbols referencing target memory	`ADD` on page 2-18
Create host-debugger symbols	`ADD` on page 2-18
Delete symbols	`DELETE` on page 2-114
Browse C++ class structure	`BROWSE` on page 2-87
Load only the symbols for a program	`LOAD` on page 2-189
Display symbols in the symbol table	`PRINTSYMBOLS` on page 2-224

Table 2-9 Program symbol manipulation commands (continued)

Description	Command
Display variable type details	`PRINTTYPE` on page 2-227
Evaluate expressions involving symbols	`CEXPRESSION` on page 2-91 `PRINTVALUE` on page 2-229
Display value of symbol every time debugger stops target	`MONITOR` on page 2-204 `NOMONITOR` on page 2-208

2.2.10 Creating and writing to files and windows

Table 2-10 shows the commands that manipulate windows.

Table 2-10 Creating files and text writing commands

Description	Command
Opening a file	`FOPEN` on page 2-166
Creating a new window	`VOPEN` on page 2-343
Clearing a window	`VCLEAR` on page 2-336
Setting the cursor position within a window	`VSETC` on page 2-345
Deleting a window	`VCLOSE` on page 2-337
Attaching a macro to a window	`VMACRO` on page 2-341
Display list of open files and windows	`WINDOW` on page 2-350
Writing text to a file or window	`FPRINTF` on page 2-169 `PRINTF` on page 2-220 Commands that support the `;`*`windowid`* or `;`*`fileid`* parameter.

2.2.11 Processor tracing

Table 2-11 shows the processor instruction tracing functions.

Table 2-11 Processor tracing commands

Description	Command
Enable and disable tracing	`TRACE` on page 2-288
Configure the trace capture logic	`ANALYZER` on page 2-28 `ETM_CONFIG` on page 2-153
Display status information	`DTRACE` on page 2-139
Set tracepoints in the program	`TRACE` on page 2-288 `TRACEDATAACCESS` on page 2-301 `TRACEDATAREAD` on page 2-307 `TRACEDATAWRITE` on page 2-313 `TRACEEXTCOND` on page 2-318 `TRACEINSTREXEC` on page 2-323 `TRACEINSTRFETCH` on page 2-328
Displaying, saving, and loading captured trace information	`TRACEBUFFER` on page 2-291

2.2.12 RTOS debugging

Table 2-12 shows the commands that are specific to OS aware connections.

Table 2-12 RTOS-specific debugging commands

Description	Command
Control OS awareness	`OSCTRL` on page 2-214
RTOS action commands	`AOS_`*`resource-list`* on page 2-32
RTOS resource commands	`DOS_`*`resource-list`* on page 2-129
Select thread in RTOS thread group	`THREAD` on page 2-287

Table 2-13 shows those commands that provide arguments or have specific behavior for OS aware connections.

Table 2-13 Debugging commands with RTOS-related features

Description	Command
Set an instruction breakpoint for a specific thread	`BREAKINSTRUCTION` on page 2-61
Stop execution at current point	`HALT` on page 2-177 `STOP` on page 2-283
Reset processor and cleanup thread states and other OS issues	`RESET` on page 2-241

Other commands can be used with OS aware connections, such as those for stepping, accessing memory and registers, and setting hardware breakpoints.

2.2.13 Miscellaneous

Table 2-14 shows the remaining functions.

Table 2-14 Miscellaneous commands

Description	Command
Open and close the Connection Control window	`CCTRL` on page 2-90
Get help on command	`HELP` on page 2-178 `DHELP` on page 2-119 `DCOMMANDS` on page 2-107
Run a command on the host operating system	`HOST` on page 2-179
Define user preferences	`OPTION` on page 2-211 `SETTINGS` on page 2-263
Force debugger to wait for a specified number of seconds	`PAUSE` on page 2-217
Force debugger to wait, or not to wait, for command to complete	`WAIT` on page 2-346
Quit debugger	`QUIT` on page 2-233

2.3 Alphabetical command reference

This section lists in alphabetical order all the commands that you can issue to RealView Debugger using the CLI. It also lists all aliases and includes cross references, where appropriate.

The following sections describe the available commands:

- *ADD* on page 2-18
- *ADDBOARD* on page 2-21
- *ADDFILE* on page 2-23
- *ALIAS* on page 2-25
- *ANALYZER* on page 2-28
- *AOS_resource-list* on page 2-32
- *ARGUMENTS* on page 2-34
- *BACCESS* on page 2-37
- *BEXECUTION* on page 2-37
- *BGLOBAL* on page 2-38
- *BINSTRUCTION* on page 2-41
- *BREAD* on page 2-42
- *BREAK* on page 2-42
- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *BROWSE* on page 2-87
- *BWRITE* on page 2-88
- *CANCEL* on page 2-89
- *CCTRL* on page 2-90
- *CEXPRESSION* on page 2-91
- *CLEARBREAK* on page 2-94
- *COMPARE* on page 2-96
- *CONNECT* on page 2-98
- *CONTEXT* on page 2-102
- *COPY* on page 2-104
- *DBOARD* on page 2-106
- *DBREAK* on page 2-106
- *DCOMMANDS* on page 2-107

- *INCLUDE* on page 2-181
- *INTRPT* on page 2-183
- *JOURNAL* on page 2-184
- *JUMP* on page 2-186
- *LIST* on page 2-187
- *LOAD* on page 2-189
- *LOG* on page 2-193
- *MACRO* on page 2-195
- *MEMMAP* on page 2-197
- *MEMWINDOW* on page 2-201
- *MMAP* on page 2-202
- *MODE* on page 2-203
- *MONITOR* on page 2-204
- *NAMETRANSLATE* on page 2-206
- *NOMONITOR* on page 2-208
- *ONSTATE* on page 2-209
- *OPTION* on page 2-211
- *PATHTRANSLATE* on page 2-215
- *PAUSE* on page 2-217
- *PRINTDSM* on page 2-218
- *PRINTF* on page 2-220
- *PRINTSYMBOLS* on page 2-224
- *PRINTTYPE* on page 2-227
- *PRINTVALUE* on page 2-229
- *PROPERTIES* on page 2-232
- *PS* on page 2-232
- *PT* on page 2-232
- *QUIT* on page 2-233
- *READBOARDFILE* on page 2-234
- *READFILE* on page 2-235
- *REEXEC* on page 2-238
- *RELOAD* on page 2-239
- *RESET* on page 2-241
- *RESETBREAKS* on page 2-243
- *RESTART* on page 2-245
- *RSTBREAKS* on page 2-247
- *RTOS action commands* on page 2-248

- *RTOS resource commands* on page 2-249
- *RUN* on page 2-250
- *SCOPE* on page 2-252
- *SEARCH* on page 2-254
- *SETFLAGS* on page 2-256
- *SETMEM* on page 2-257
- *SETREG* on page 2-259
- *SETTINGS* on page 2-263
- *SHOW* on page 2-269
- *SINSTR* on page 2-270
- *SM* on page 2-270
- *SOINSTR* on page 2-270
- *SOVERLINE* on page 2-270
- *SR* on page 2-270
- *STDIOLOG* on page 2-271
- *STEPINSTR* on page 2-273
- *STEPLINE* on page 2-275
- *STEPOINSTR* on page 2-278
- *STEPO* on page 2-280
- *STOP* on page 2-283
- *TEST* on page 2-285
- *THREAD* on page 2-287
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328
- *UNLOAD* on page 2-333
- *UP* on page 2-335
- *VCLEAR* on page 2-336
- *VCLOSE* on page 2-337
- *VERIFYFILE* on page 2-338
- *VMACRO* on page 2-341
- *VOPEN* on page 2-343

- *VSETC* on page 2-345
- *WAIT* on page 2-346
- *WARMSTART* on page 2-347
- *WHERE* on page 2-348
- *WINDOW* on page 2-350
- *WRITEFILE* on page 2-351

2.3.1 ADD

The `ADD` command creates a symbol and adds it to the debugger symbol table.

Syntax

`ADD [`*`type`*`]` *`symbol_name`* `[&`*`address`*`] [=`*`value`* `[,`*`value`*`]...]`

where:

`type` One of the following data types:

`int`	The symbol represents a location holding a four byte signed integer value. This is the default type of symbols.
`char`	The symbol represents a location holding a one byte value.
`short`	The symbol represents a location holding a two byte value.
`long`	The symbol represents a location holding a four byte value.
`long long`	The symbol represents a location holding an 8-byte value.

You can use these types together with `*` and `[]` to indicate pointer and array variables, using the C language syntax.

You can also create symbols with *`type`* **`float`** or **`double`**, but you cannot initialize them with a value in the `ADD` statement.

You can create references to existing instances of the following types:

`struct`	The symbol is an instance of, or a pointer to, a C structure.
`enum`	The symbol is an instance of, or a pointer to, a C enumeration.
`union`	The symbol is an instance of, or a pointer to, a C union.

You cannot create new enumerations, structures, or unions. You cannot initialize complete structures at once, although you can individually assign values to the members with `CEXPRESSION`.

If the symbol is an array, then the array size must be specified after to the symbol name within in square brackets. You can define multidimensional arrays by appending several bracketed array dimensions.

`symbol_name` Is the name of the symbol being added. The name must start with an alphabetic character or an underscore, optionally followed by alphabetic or numeric characters or underscores. The symbol name must not already exist (when appropriate, use `DELETE` on page 2-114 to remove a symbol).

`address` Is the address in target memory that is referred to by this debugger symbol. If you do not specify an address, the new debugger symbol refers to a location in debugger memory, and is not available to code running on the target.

value Is the initial value of the added symbol. You can use:

- integer values corresponding to the C types **int**, **char**, **short**, **long** or **long long**
- pointers to integers in target memory
- strings in double quotes, matching the character array type, **char**[*n*], but not **char** *
- a list of values separated by a comma.

If the symbol type is a pointer, an assigned value must be the address of the value on the target.

You can initialize array symbols using multiple *value* arguments. For example:

```
add char names[3][2] ="aa", "bb"
print names[1]
"bb...
```

The ... after bb indicates that there is no terminating NUL for the string, in this case because each element of the array is only 2 characters in size.

The value is loaded into the memory location referred to by the symbol. If value is not specified, the symbol is set to zero in debugger memory but is not given a value in target memory.

Floating-point values are not recognized.

Description

The ADD command adds a symbol to the debugger symbol table for the current connection. You cannot add a symbol without a connection, but you do not have to have loaded an executable image file. The symbol survives an executable image being reloaded (**File → Reload Image to Target**) but is destroyed if the target is disconnected and then reconnected or another, different, image is loaded.

You can remove a symbol defined using ADD by using the DELETE command, and you can modify its value using CEXPRESSION.

Rules

The following rules apply to the use of the ADD command:

- ADD runs asynchronously unless in a macro.
- The specified symbol must not exist at the time it is added.
- To change the symbol type, delete the symbol and then add it again.

- When initializing symbols, the size of the symbol is used, not the size of the type of value supplied. In particular, the size of a `char` array is not determined by the string used to initialize it.
- If a char array is created, for example using `ADD char namearray[n]`, it is filled with the initial value.
- If there is a runtime error in the initial value, the symbol is still created. You can then assign the correct value with the `CEXPRESSION` command, or you can delete the symbol and add it again with a legal initial value.

Examples

The following examples show how to use `ADD`:

`add mysymbol =3`	Adds a new symbol called `mysymbol` of type **`int`** to the debugger symbol table.
`add char *xyz`	Adds a new symbol called `xyz` to the debugger symbol table. The new symbol is of character pointer type and has an initial value of zero.

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *DELETE* on page 2-114.

2.3.2 ADDBOARD

The `ADDBOARD` command creates or modifies a board file entry, but the changes are not persistent across debugger sessions.

Syntax

```
ADDBOARD name[:port] ="description", "manufacturer" [,"path_translation"]
[,"io_port"]
```

where:

name[:*port*] The name of the board file entry that you are adding or modifying and, optionally, a port. If you specify a port, then you must enter the name and port in double quotes, for example:

```
"name:port"
```

description A description that is to appear in the Description column of the Connection Control window.

manufacturer Specify a manufacturer name in *MMM-VVV-CCC* format:

MMM	A three-letter code to identify the manufacturer, for example `ARM`.
VVV	A vehicle identifier of up to three characters.
CCC	Identifies the connection type, and must be: **USB** Universal Serial Bus

For example, "`ARM-ARM-USB`".

path_translation

(Optional). The path translation between the host and the target, and is in the format:

```
hpath=tpath
```

where `hpath` is the path on the host and `tpath` is the path on the target. Both `hpath` and `tpath` can be a comma-separated list.

Target can be `'-'` which means a grab of a process with no path prepends the host path.

io_port (Optional). An I/O port number.

Description

The ADDBOARD command enables you to add or modify a board definition for a connection. The changes are made in debugger memory, so that you do not have to change the board files for the connection.

The command does not modify the file on disk.

Examples

The following command adds a target board entry called MyBoard under ARM-A-RR:

```
addboard "MyBoard" ="My board connection","ARM-A-RR"
```

See also

The following commands provide similar or related functionality:

- *CONNECT* on page 2-98
- *DELBOARD* on page 2-113
- *PATHTRANSLATE* on page 2-215.

2.3.3 ADDFILE

The `ADDFILE` command adds the named file to the executable image file list but does not load it. You can optionally empty the list before adding the new filename.

Syntax

```
ADDFILE [,auto] =filename [=string,...]
```

where:

`auto`
: Specifies that only one added file is permitted for each process or processor. Any previously added file is removed when the specified file is added.

`filename`
: The name of the file to be added. You must use double quotes around the filename if it contains any spaces.

  You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myimages`, you can specify:

  ```
  adsfile ="$MYPATH\\myimage.axf"
  ```

`string`
: The target pathname, for example, an RTOS loader.

Description

The RealView Debugger executable file list contains the names of the files containing the target code for your application. Normally this contains a single linker output file, for example `dhry.axf` and, in this case, you use the `LOAD` and `RELOAD` commands as required.

However, when the application is more complex it is sometimes easier to create it as several files, for example one file for the *Operating System* (OS) and one for each major process. In these cases, you use the `ADDFILE` and `RELOAD`, or the `ADDFILE` and `LOAD/A` commands, to manipulate the files that are loaded onto the target.

To load the files on the file list use `RELOAD`, described on page 2-239.

Examples

The following example removes any existing files from the executable file list and loads `dhry.axf` into it. The reload command then transfers the executable contents of `dhry.axf` to the target and sets the processor PC to the image entry point:

```
addfile,auto =c:\source\debug\dhry.axf
reload
```

This is the same as:

```
load c:\source\debug\dhry.axf
```

This example loads the file `dhry.axf` into the file list, removing any existing files. It then adds the file `kernel.axf` to the file list. The reload command transfers the executable contents of both files to the target and sets the PC to the entry address of the last executable loaded, in this case that of `kernel.axf`.

```
addfile,auto =c:\source\dhry\debug\dhry.axf
addfile =c:\source\OS\debug\kernel.axf
reload
```

See also

The following commands provide similar or related functionality:

- *DELFILE* on page 2-116
- *DTFILE* on page 2-137
- *LOAD* on page 2-189
- *RELOAD* on page 2-239
- *UNLOAD* on page 2-333.

2.3.4 ALIAS

The `ALIAS` command is used to manipulate command aliases. Aliases are new debugger commands constructed from (optionally, parts of) existing debugger commands or macros.

Syntax

`ALIAS [alias_name [=[definition]]]`

where:

alias_name	Names your new debugger command. This name is accepted as a legal debugger command name.
	An optional asterisk `*` embedded in the name indicates that the parts of the name that follow are not required, so your command can be abbreviated.
definition	Defines the replacement string that is substituted in place of *alias_name* whenever *alias_name* is invoked.
	The definition normally contains macro invocations or debugger internal commands, or parts of such commands. However, any string of legal debugger characters is accepted.
	Using `$*` within a definition inserts the command-line parameters to the alias in the expansion. By default, parameters are appended to the alias when command expansion occurs.

Description

The `ALIAS` command can create, list, or delete new debugger commands. The building blocks are existing debugger commands and macros and, optionally, specific parameters. You can use `ALIAS` to define either:

- a new name (for example, one that is shorter or easier to remember) for an existing command
- a command that defines fixed parameters for an existing command.

`ALIAS` can only substitute one command for another. If you require a multiple command alias, use the `MACRO` command instead.

Enter `ALIAS` without parameters to display a list of the defined alias commands in the order in which they were added.

You can name your alias using almost any sequence of letters or numbers. However, when a command is entered the debugger searches for internal debugger commands before it searches for aliases. Therefore, you must ensure that you do not use an alias name that is the same as an internal debugger command. The name priorities are as follows:

1. Debugger internal command, or defined abbreviation of command
2. Defined alias names, and the defined abbreviations of alias names
3. Macro names.

You can place alias command arguments in a specific position in the expanded debugger command by inserting the sequence `$*` where the parameters to the command alias must appear.

Rules

The following rules apply to the use of the `ALIAS` command:

- `ALIAS` runs asynchronously unless it is called within a macro.
- *`alias_name`* must not exist at the time it is added. To change the definition of an alias, first define the alias equal to the nothing (`alias nm=`) to delete it and then add it again.
- If a debugger command has the same name as an alias, the debugger command is the one that is executed.
- Alias names are always matched before macros names.
- If two alias abbreviations or an alias and an abbreviation match, the first alias added during the current session is always used.
- An alias definition must be defined in terms of predefined debugger commands or macro names.
- An alias definition can reference debugger commands and macros.

Examples

The following examples show how to use `ALIAS`:

```
alias showf*ile =dtfile ;99
```

Defines a command called `SHOWFILE` that can be abbreviated to `SHOWF`, that is equivalent to the `DTFILE` command with its output directed to window number `99`.

```
alias dub =dump /b
```

Defines a command called DUB, with no abbreviation, that expands to the DUMP command in byte mode (/b).

```
dub 0x20
```

Calls the alias dub to print out memory in bytes from address 0x20. This alias invocation is exactly the same as typing:

```
dump /b 0x20
```

```
alias bpc =breakexecution,continue,message:{Break} $* ;DoCheck()
```

Defines a command called BPC, with no abbreviation, that expands to the breakexecution command with specific parameters and trigger macro DoCheck(). It must be invoked with the address to break at as a parameter:

```
bpc \MAIN_C\#15
```

This is equivalent to typing the command:

```
breakexecution,continue,message:{Break} \MAIN_C\#15 ;DoCheck()
```

See also

The following commands provide similar or related functionality:

- *BREAKEXECUTION* on page 2-52
- *DTFILE* on page 2-137
- *MACRO* on page 2-195.

2.3.5 ANALYZER

The `ANALYZER` command controls the configuration of the trace logic analyzer.

Syntax

```
ANALYZER {[,disable]|[,enable]}

ANALYZER
{[edit_properties]|[,map_log_phys]|[,triggers]|[,connect]|[,set_size]}

ANALYZER {[,clear]|[,clear_triggers]}

ANALYZER
{[,before]|[,around]|[,after]|[,stop_on_trigger]|[,continue_on_trigger]}

ANALYZER {,full_stop|,full_ignore|,full_ring}

ANALYZER {,collect_all|,collect_flow}

ANALYZER {,dataonly|,addronly|,fulltrace}

ANALYZER {,disconnect}

ANALYZER {,auto_off|,auto_instronly|,auto_dataonly|,auto_both}

ANALYZER {,mode_continuous|,mode_trigger}
```

where:

`disable`	Disable tracing.
`enable`	Enable tracing.
`edit_properties`	When not connecting to an ARM® ETM-enabled processor, this is the equivalent of the **Edit → Configure Analyzer Properties...** option on the Analysis window main menu. To configure an ETM use `ETM_CONFIG`. (Deprecated)
`map_log_phys`	The equivalent of the **Edit → Physical to Logical Address Mapping...** option on the Analysis window main menu. Not available with an ARM ETM-enabled processor.
`triggers`	The equivalent of the **Edit → Set/Edit Event Triggers** option on the Analysis window main menu. Not available with an ARM ETM-enabled processor.

`connect`	The equivalent of the **Edit → Connect Analyzer** option on the Analysis window main menu. Not available with an ARM ETM-enabled processor because an ARM ETM is automatically connected.
`set_size=(`*`n`*`)`	Enables you to set the trace buffer size. The equivalent of the **Edit → Set Trace Buffer Size...** option on the Analysis window main menu. If the value is specified in the command it is used, otherwise display the Set Trace Buffer Size dialog and set the value from that.
`clear`	Clear the captured trace buffer.
`clear_triggers`	Clear any triggers set using an `ANALYZER,triggers` command. Not available with an ARM ETM-enabled processor.
`before`	Capture data before the trigger, that is, 100% before, 0% after.
`around`	Capture data around the trigger, that is, 50% before, 50% after.
`after`	Capture data after the trigger, that is, 0% before, 100% after.
`stop_on_trigger`	Stop the processor when a trigger point is reached. This option is only applicable to the ARM ETM.
`continue_on_trigger`	Continue program execution across trigger points. This option is only applicable to the ARM ETM.
`full_stop`	Stop the processor and put it into debug state when the trace buffer is full. Not available with an ARM ETM-enabled processor.
`full_ignore`	Stop collecting trace information when the trace buffer is full, but let the processor continue running. Not available with an ARM ETM-enabled processor.
`full_ring`	Continue collecting trace information when the trace buffer fills by discarding the oldest trace information, treating the buffer as a ring. This is the only option available for the ARM ETM.
`collect_all`	Store all trace the information generated. Not available with an ARM ETM-enabled processor.
`collect_flow`	Store only flow-control trace information. Cannot be changed for an ARM ETM-enabled processor because normal ETM operation is a variant of this that includes some additional synchronization points.

`dataonly`	Trace only data bus transfers.
`addronly`	Trace only address bus transfers.
`fulltrace`	Trace both data and address bus transfers.
`disconnect`	Disconnects the Analysis window.
`auto_off`	Disables automatic tracing.
`auto_instronly`	When no tracepoints are set, captures trace information only for executed instructions.
`auto_dataonly`	When no tracepoints are set, captures trace information only for data accesses. This is supported only by ETM v3.
`auto_both`	When no tracepoints are set, captures trace information for both executed instructions and data accesses.

Description

The `ANALYZER` command, and the `ETM_CONFIG` command, enables you to control the configuration of your trace capture analyzer.

——— **Note** ———

Because trace analyzer capabilities and implementations vary, some of the qualifiers provided by the `ANALYZER` command are not available on some of the trace targets supported by RealView Debugger. Operation of the ARM ETM is controlled in more depth with the `ETM_CONFIG` command.

———————————

For more information analysis and tracing, see the *Embedded Trace Macrocell Specification* and the chapter that describes tracing in the *RealView Developer Kit v2.2 Extensions User Guide*.

The options are split into several groups:

- Options `config`, `edit_properties`, `map_log_phys`, `triggers`, and `set_size` display a GUI dialog that enables you to configure the associated trace component.

  ——— **Note** ———

  These options are not available in the headless debugger.

  ———————————

- The clear option acts on the trace capture buffer. See also the `TRACEBUFFER` command.

- Options `before`, `around`, `after`, `clear_triggers`, `stop_on_trigger`, and `continue_on_trigger` enable you to control the relative location of the trace trigger within the trace buffer and the effect of the trigger. See the `TRACE`, `TRACEINSTREXEC`, `TRACEDATAACCESS` and similar commands for control of tracepoint location in target memory.
- Options `full_stop`, `full_ignore`, and `full_ring` enable control over the behavior of the trace buffer when it becomes full.
- Options `collect_all` and `collect_flow` enable control of the trace data collection strategy. Collecting all bus transactions provides the benefit of following everything that is happening without recourse to external information, but conversely requires a very high bandwidth trace port. Collecting only bus transactions that change the flow of control provides most of the important information if you also have access to an accurate memory image.

Examples

The following examples show how to use `ANALYZER`:

`ANALYZER,set_size=500`

Set the trace buffer size to 500 records, if this action is supported by the logic analyzer you are using.

`ANALYZER,full_ring,around`

Set the logic analyzer to capture trace information around the defined trigger point, using the trace buffer in ring mode so that it cannot overflow.

See also

The following commands provide similar or related functionality:

- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAWRITE* on page 2-313
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.6 AOS_resource-list

Performs an action on an object chosen from the RTOS resource list.

Syntax

`AOS_resource-list ,qualifier [=value]`

where:

resource-list

Specifies the resource list.

qualifier	Specifies the action, that is *action-name*[:*value*].
value	Identifies an object in the specified resource list.

Description

The `AOS_resource-list` command performs an action on an object chosen from the RTOS resource list. The *resource-list* and *qualifier* depend on the RTOS you are using.

You can get a list of these commands using the `DCOMMANDS` command. To do this, and save the commands to the file `rtoscmds.txt`, enter:

```
fopen 100,'c:\path\rtoscmds.txt'
dcommands all ;100
vclose 100
```

You can also determine the commands from the Resource Viewer window:

- *resource-list* is determined by the tab you select in the Resource List, with the exception of the **Conn** tab
- *qualifier* is determined by right clicking on an object in the selected tab of the Resource List.

You might want to log your use of the Resource Viewer window to determine the CLI commands you can use with your RTOS. See *LOG* on page 2-193 for details. You can then modify the log file, and use it as a command script, see *INCLUDE* on page 2-181.

See the chapter that describes RTOS support in the *RealView Developer Kit v2.2 Extensions User Guide* for details on using the Resource Viewer window, and the interaction of RTOS actions and RealView Debugger.

Examples

The following examples show how to use `AOS_`*`resource-list`*:

`aos_thread_list,suspend = 0x39d8`

Suspends the thread `0x39d8`.

`aos_timer_list,set:100 = 0x15260`

Sets the value of the timer `0x15260` to `100`.

`aos_timer_list,deactivate = timer_1`

Deactivates the timer `timer_1`.

See also

The following commands provide similar or related functionality:

- *DOS_resource-list* on page 2-129
- *OSCTRL* on page 2-214

2.3.7 ARGUMENTS

The `ARGUMENTS` command enables you to specify the command-line arguments for the application. These are used for each subsequent run on this connection.

Syntax

```
ARGUMENTS [{,delete | ,prompt}]
```

```
ARGUMENTS [,default] string
```

where:

`delete`	Delete the currently set `ARGUMENTS` list, so the `argv` list for the next run of a program is only the program filename.
`default`	Make the defined arguments the default, so they apply to new connections created in this session.
`prompt`	Display a dialog to prompt you for the arguments when the `ARGUMENTS` command is executed.
`string`	Defines the command line that the application sees when it inspects the `argv[]` array, or equivalent.

Description

The `ARGUMENTS` command enables you to specify arguments that the target application might require when it starts execution. The specified string is made available to ARM applications through the semihosting mechanism. Any previous argument definition is overwritten.

If a literal double-quote character is required in the arguments, you must escape it using the backslash character and embed it in single quotes, for example:

```
ARGUMENTS "-f '\"my file.c\"'
```

If you enter this command without any parameters, the current argument definition is displayed.

——— **Note** ———

You can also specify arguments as part of the `LOAD` command.

If you unload an image that requires arguments, any arguments defined with this command are not used by the image when you next reload it. After reloading the image, enter this command again to specify the arguments before running the image.

————————

Examples

The following examples show how to use `ARGUMENTS`:

```
ARGUMENTS "-f file.c -o file.o"
```

Sets the command line so that, if the line is parsed in the normal way by `_main()`, the `argv[]` array contains:

argv[0]	target program filename, for example: `com.axf`
argv[1]	`-f`
argv[2]	`file.c`
argv[3]	`-o`
argv[4]	`file.o`
argv[5]	`NULL`

```
ARGUMENTS "-f '\"my file.c\"' -o '\"my file.o\"'"
```

Sets the command line so that, if the line is parsed in the normal way by `_main()`, the `argv[]` array contains:

argv[0]	target program filename, for example: "`com.axf`"
argv[1]	`-f`
argv[2]	`"my file.c"`
argv[3]	`-o`
argv[4]	`"my file.o"`
argv[5]	`NULL`

```
load /pd/r 'com.axf;;-f file.c -o file.o' go arguments "-f '\"my file.c\"' -o '\"my
file.o\"'" restart go
```

Changes the arguments without unloading the image. Table 2-15 shows the argument assignments in the original `LOAD` command, and the new assignments specified by the `ARGUMENTS` command.

Table 2-15 Changed argument assignments

Argument	Original value	New value
`argv[0]`	`com.axf`	`com.axf`
`argv[1]`	`-f`	`-f`
`argv[2]`	`file.c`	`"my file.c"`

Table 2-15 Changed argument assignments

Argument	Original value	New value
`argv[3]`	`-o`	`-o`
`argv[4]`	`file.o`	`"my file.o"`
`argv[5]`	`NULL`	`NULL`

See also

The following commands provide similar or related functionality:

- *GO* on page 2-173
- *LOAD* on page 2-189
- *RESTART* on page 2-245.

2.3.8 BACCESS

`BACCESS` is an alias of `BREAKACCESS` (see page 2-43).

2.3.9 BEXECUTION

`BEXECUTION` is an alias of `BREAKEXECUTION` (see page 2-52).

2.3.10 BGLOBAL

The `BGLOBAL` command enables or disables global breakpoints, also called *processor exceptions*.

Note

This command overrides the settings in the Connection Properties of the current connection. However, if you disconnect and reconnect then the settings in the Connection Properties are applied to the connection.

Syntax

```
BGLOBAL {,enable | ,disable} name [;macro-call]
```

```
BGLOBAL ,gui [;macro-call]
```

where:

request If specified, must be one of the following:

`enable`	Enable the specified global breakpoint.
`disable`	Disable the specified global breakpoint.
`gui`	Display a list box that enables you to select a global breakpoint to enable or disable.

Note

This qualifier has no effect in the headless debugger.

name Identifies the global breakpoint to be enabled or disabled. See *Compatibility* on page 2-39 for a list of supported names.

macro-call Specifies a macro and any parameters it requires. This macro is run when a global breakpoint is triggered.

If the macro returns a nonzero value, execution continues. If the macro returns a zero value, or if you do not specify a macro, target execution stops and the debugger waits in command mode.

Description

The `BGLOBAL` command enables or disables global breaks. A global breakpoint is a processor event that can cause execution to halt in any application using this connection. For example, taking an undefined instruction trap might be a global breakpoint. The list of possible global breakpoint events is defined by a combination of the target processor and the target vehicle. For more information on the meaning of the processor exceptions see the processor architecture manual.

Compatibility

The supported events are determined in part by the currently connected processor type:

RealView ICE and RVI-ME connections to ARM processors

The possible events are the exception types supported by the processors on RVI and RVI-ME connections. The supported *name*s are:

reset	The RESET exception.
undef	The undefined instruction exception.
SWI	The software interrupt (SWI) exception.
prefetch abort	The prefetch abort (instruction memory read abort) exception. You must use double quotes to specify this name, for example: `bglobal,enable "prefetch abort"`
data abort	The data abort (data memory read or write abort) exception. You must use double quotes to specify this name, for example: `bglobal,enable "data abort"`
IRQ	The interrupt request exception.
FIQ	The fast interrupt request exception.

Examples

The following example disables debugger interception of the ARM architecture SWI exception, so that an application can process SWI exceptions itself:

```
bglobal,disable SWI
```

This example enables debugger interception of the ARM architecture `UNDEF` exception, so that if the application starts executing data literals (the usual reason for unintentionally executing an undefined instruction) you can find out why:

```
bglobal,enable undefined
```

Some processor exceptions interact with other debugger functions. For example, the ARM SWI exception is used by the ARM Semihosting interface.

See also

The following command provides similar or related functionality:

- *GO* on page 2-173.

2.3.11 BINSTRUCTION

`BINSTRUCTION` is an alias of `BREAKINSTRUCTION` (see page 2-61).

2.3.12 BREAD

BREAD is an alias of BREAKREAD (see page 2-69).

2.3.13 BREAK

BREAK is an alias of BREAKINSTRUCTION (see page 2-61).

2.3.14 BREAKACCESS

The `BREAKACCESS` command sets a hardware breakpoint that triggers when specified memory locations are accessed, either by a memory read or a memory write.

See the chapter that describes working with breakpoints in the *RealView Developer Kit v2.2 Debugger User Guide* for full details on the use of breakpoints in RealView Debugger.

Syntax

```
BREAKACCESS [,qualifier...] {address | address_range} [;macro_call]
```

where:

`qualifier` Is a list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers* on page 2-44.

`address | address_range`

Specifies a single address in target memory, or an address range. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

`macro_call` Specifies a macro and any parameters it requires. This macro runs when the access breakpoint is triggered. The macro is treated as being specified last in the qualifier list.

If the macro returns a nonzero value, or you specified `continue` in the qualifiers, execution continues. If the macro returns a zero value, or if you do not specify a macro, target execution stops and the debugger waits in command mode.

The macro argument symbols are interpreted when the breakpoint is specified and so they must be in scope at that point, or you must explicitly qualify them.

Description

`BREAKACCESS` is used to set or modify memory access breakpoints. Access breakpoints trigger when one or more specified memory addresses are read from or written to. If the command has no arguments, it behaves like `DTBREAK` on page 2-135, listing the current breakpoints (see *List of qualifiers* on page 2-44).

Hardware address breakpoints can use other hardware tests in association with the address test, such as trigger inputs and outputs, hardware pass counters, and *and-then*, or chained, tests.

All breakpoints can add on-host qualifiers, for example expressions, macros, C++ object tests, and pass counters. All address breakpoints can include on-host actions including: counters, timing (with hardware assist), update of specified windows, enabling or disabling other breakpoints, and the starting and stopping other processors or threads.

When a hardware data access breakpoint is hit on the target, the following sequence of events occurs:

1. The debugger or the hardware associates the event with a specific debugger breakpoint ID.
2. If the breakpoint has a software pass count associated with it, the count is updated.
3. The conditions for this breakpoint, if any, are tested in the order specified on the command line. If any condition is False, target execution resumes with the instruction at the breakpointed location. Macros specified with the `macro:` qualifier or the `;`*`macro_call`* argument are run in this phase.
4. If the breakpoint has actions associated with it (for example, using `timed` to note the time the breakpoint occurred) these actions are run, in the order specified on the command line.
5. If the qualifiers include `continue`, target execution resumes with the instruction at the breakpointed location. If not, the debugger updates the state of the GUI and waits for a command, leaving the application halted.

List of qualifiers

The list of qualifiers depends on the processor and vehicle and so the GUI does not present things that do not make sense. The command handler generates an error if a specific combination is invalid for a specific processor or vehicle, but this is determined when you issue the command.

To set up a conditional breakpoint, use the following condition qualifiers:

- `macro` (or `;`*`macro-call`*)
- `obj`
- `passcount`
- `when`
- `when_not`.

To specify actions to be performed when a breakpoint triggers, use the following action qualifiers:

- `continue`
- `message`

- `sample` (not supported in this release)
- `timed`
- `update`.

The possible qualifiers are:

`append:(n)`

Instead of creating a new breakpoint, append the qualifiers specified with this command to an existing breakpoint with breakpoint ID number *n*.

Note

You cannot use `append` to change the breakpoint address or to create chained breakpoints.

`continue`

Execution continues when the breakpoint is triggered and no breakpoint details are displayed. Any specified action qualifiers are still performed, depending on the results of any condition qualifiers.

`context:{context}`

Sets the context for other expressions in this breakpoint command to the value of context. This provides an alternative to specifying the complete context for every symbol. For example:

```
BREAKACCESS,context:{DHRY_1},when:{Int_1_Loc>0} #152
```

This causes a breakpoint to be set at line 152 of `dhry_1.c` that is triggered only when the variable `status` defined in `dhry_1.c` is greater than zero. The alternative form is:

```
BREAKACCESS,when:{Int_1_Loc>0} \DHRY_1\#152
```

Note

When using this alternative form of the command, the filename `DHRY_1` must be uppercase.

`data_only`

The breakpoint is triggered if a data value, specified using `hw_dvalue`, is detected by the debug hardware on the processor data bus.

`gui`

If an error occurs when executing the command or when the breakpoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

Note

This qualifier has no effect in the headless debugger.

`hw_ahigh:(`*n*`)`	Specifies the high address for an address-range breakpoint. The low address is specified by the standard breakpoint address. This facility is not supported by ARM EmbeddedICE® macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1000-0x1200`: `BREAKACCESS,hw_ahigh:0x1200 0x1000`
`hw_amask:(`*n*`)`	Specifies the address mask value for an address-range breakpoint. Addresses that match the standard breakpoint address when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1FA00-0x1FA0F`: `BREAKACCESS,hw_amask:0xFFFF0 0x1FA00`
`hw_dvalue:(`*n*`)`	Specifies a data value to be compared to values transmitted on the processor data bus. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for the data value `0x400`: `BREAKACCESS,hw_dvalue:0x400`
`hw_dhigh:(`*n*`)`	Specifies the high data value for a data-range breakpoint. The low data value is specified by the `hw_dvalue` qualifier. This facility is not supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x00-0x18`: `BREAKACCESS,hw_dvalue:0x0,hw_dhigh:0x18 0x1000`
`hw_dmask:(`*n*`)`	Specifies the data value mask value for a data-range breakpoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x400-0x4FF`: `BREAKACCESS,hw_dvalue:0x400,hw_dmask:0xF00 0x1FA00`

hw_passcount:(*n*)
: Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. This qualifier differs from `passcount` only in that it is implemented in hardware. *n* is limited to a 32-bit value by the debugger, but might be much more limited by the target hardware, for example to 8 or 16 bits.

You can combine hardware and software pass counts to achieve higher count values. If you define both hardware and software pass counts:

1. when the hardware count reaches zero, the software count is decremented
2. when the software count reaches zero, the breakpoint triggers.

hw_and:{[then-]*id*}
: Perform an *and* or an *and-then* conjunction with an existing breakpoint. For example, `hw_and:{2}`, or `hw_and:{then-2}`, where *2* is the breakpoint id of another breakpoint.

In the *and* form, the conditions associated with both breakpoints are chained together, so that the action associated with the second breakpoint are performed only when both conditions simultaneously match.

In the *and-then* form, when the condition for the first breakpoint is met, the second breakpoint is enabled but the program is not yet stopped. When the second breakpoint condition is matched, the actions associated are performed. At this point, unless the `continue` qualifier is specified in the second breakpoint, the program stops.

The *id* is one of:

- the breakpoint list index of an existing breakpoint
- `next` for the next breakpoint specified for this connection
- `prev` for the last breakpoint specified for this connection.

When a breakpoint is set with `hw_and:next`, and another is set with `hw_and:prev`, these two breakpoints are chained. For example:

```
BREAKACCESS,hw_and:next,hw_dvalue:1
    @copyfns\\COPYFNS\mycpy\append
BREAKACCESS,hw_and:prev @copyfns\\COPYFNS\mycpy\
```

If you clear the breakpoint that has the ID `next`, then both breakpoints are cleared.

If you clear the breakpoint that has the ID `prev`, then only that breakpoint is cleared.

Debugger internal handle numbers are not available to users to identify breakpoints.

`hw_in:{}` In trigger tests. The string that follows matches hardware-supported input tests, per vehicle and processor, as a list of names or a value.

`hw_out:{s}` Not supported in this release.

`hw_not:{s}` Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *s* can be set to:

`addr`	Invert the breakpoint address value.
`data`	Invert the breakpoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to break when a data value does not match a mask, you can write:

```
BREAKACCESS,hw_not:data,hw_dmask:0x00FF ...
```

The break commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
BREAKACCESS,hw_not:addr 0x10040
```

This means to break at any address other than `0x10040`. This example is probably not useful.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *or* and *nand-then* conditions.

This facility is not supported by ARM EmbeddedICE macrocells.

`macro:{`*`MacroCall(arg1,arg2)`*`}`

The triggering of the breakpoint results in the specified macro being executed. Any program variables or functions must be in scope at the time the breakpoint request is entered, or the names must be fully qualified. You must include the braces `{` and `}`.

<u>`mess`</u>`age:{"$`*`windowid | fileid`*`$`*`message`*`"}`

Triggering of the breakpoint results in *`message`* being output. Prefixing *`message`* with `$`*`windowid | fileid`*`$` enables you to write the message text to a user-defined window or file. See *Window and file numbers* on page 1-5 for details. For example:

```
BREAKACCESS,message:{"$100$this is a message"}
```

`modify:(`*`n`*`)`

Instead of creating a new breakpoint, modify the breakpoint with breakpoint ID number *`n`* by replacing the address expression and the qualifiers of the existing breakpoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained breakpoint to a chained breakpoint. However, you can modify a chained breakpoint with any other qualifier and also change the address expression.

`obj:(`*`n`*`)`

This condition is True if the argument *`n`* matches the C++ object pointer, normally called `this`.

`passcount:(`*`n`*`)`

Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. If you specify this in the middle of a sequence of break conditions, those specified before the passcount are processed whether or not the count reaches zero. The conditions specified afterwards are run only when the count reaches zero.

There is a hardware passcount qualifier available, `hw_passcount`, for debug hardware that supports it.

Note

If a breakpoint uses a `passcount`, the counting is performed on the host, and so program execution stops briefly every time the breakpoint is hit, even when the count has not been reached.

`register:`*`expression`*

The breakpoint is triggered if the value stored in the specified memory-mapped register is accessed in any way. The register is identified by *expression*. For example:

```
BREAKACCESS,register:PR1
```

or

```
BREAKACCESS,register:@PR1
```

Note

You can only specify registers that are defined in a Board/Chip Definition (`.bcd`) file, which you must have assigned to the connection.

`sample`

Not supported in this release.

`size:`*`n`*	Set the size of the breakpoint to either 16 or 32 bits. For example: `BREAKACCESS,size:32 0x10040` Use this qualifier if no debug information is available for your image. By default, RealView Debugger sets a 32-bit breakpoint.
`timed`	Triggering records the time, in whatever units the debug hardware chooses, from the last reset of time. The time can be in nano-seconds, micro-seconds, processor cycles, instructions, or units. See the documentation for the hardware interface for more information. The recorded times are displayed in the Resource Viewer window and in the breakpoint information for this breakpoint. The timed qualifier can be used for simple profiling, or for a measure of specific response times. If you use `timed` and `continue`, the debugger keeps a log of times for each break.
`update:{"`*`name`*`"}`	Update the named windows, or all windows, by reading the memory and processor state when the breakpoint triggers. You can use the name `all` to refresh all windows, or a name specified in the title bar of the window. This qualifier enables you to get an overview of the process state at a particular point, without having to manually restart the process at each break. The update still takes a significant period of time, and so this method is unsuitable as a non-intrusive debugging tool.
`when:{`*`condition`*`}`	The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to True. ——— **Note** ——— Using a macro as an argument to `when`, reverses the sense of the return value from the macro. ———————————
`when_not:{`*`condition`*`}`	The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to False.

Alias

`BACCESS` is an alias of `BREAKACCESS`.

See also

The following commands provide similar or related functionality:

- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *CLEARBREAK* on page 2-94
- *DTBREAK* on page 2-135
- *ENABLEBREAK* on page 2-150
- *VMACRO* on page 2-341.

2.3.15 BREAKEXECUTION

The `BREAKEXECUTION` command sets an execution breakpoint that enables ROM-based breakpoints by using the hardware breakpoint facilities of the target.

See the chapter that describes working with breakpoints in the *RealView Developer Kit v2.2 Debugger User Guide* for full details on the use of breakpoints in RealView Debugger.

Syntax

```
BREAKEXECUTION [,qualifier...] expression [;macro-call]
```

where:

qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers* on page 2-53.

expression Specifies the address at which the breakpoint is placed. For an unqualified breakpoint, this is the address at which program execution is stopped.

macro-call Specifies a macro and any parameters it requires. The macro runs when the breakpoint is triggered and before the instruction at the breakpoint is executed. The macro is treated as being specified last in the qualifier list.

If the macro returns a nonzero value, or you specified `continue` in the qualifiers, execution continues. If the macro returns a zero value, or if you do not specify a macro, target execution stops and the debugger waits in command mode.

The macro argument symbols are interpreted when the breakpoint is specified and so they must be in scope at that point, or you must explicitly qualify them.

Description

This command is used to set or modify instruction address breakpoints. Address breakpoints include breakpoints set by patching special instructions into the program and hardware that tests the address and data values. If the command has no arguments, it behaves like `DTBREAK` on page 2-135, listing the current breakpoints.

Hardware address breakpoints can use other hardware tests in association with the address test, such as trigger inputs and outputs, hardware pass counters, and *and-then*, or chained, tests.

All breakpoints can add on-host qualifiers, for example expressions, macros, C++ object tests, and pass counters. All address breakpoints can include on-host actions including counters, timing (with hardware assist), update of specified windows, enabling or disabling other breakpoints, and starting and stopping other processors or threads (see *List of qualifiers*).

When a hardware breakpoint instruction is hit on the target, the following sequence of events occurs:

1. The debugger or the hardware associates the event with a specific debugger breakpoint ID.
2. If the breakpoint has a software pass count associated with it, the count is updated.
3. The conditions for this breakpoint, if any, are tested in the order specified on the command line. If any condition is False, target execution resumes with the instruction at the breakpointed location. Macros specified with the `macro:` qualifier or the `;`*`macro_call`* argument are run in this phase.
4. If the breakpoint has actions associated with it (for example, using `timed` to note the time the breakpoint occurred) these actions are run, in the order specified on the command line.
5. If the qualifiers include `continue`, target execution resumes with the instruction at the breakpointed location. If not, the debugger updates the state of the GUI and waits for a command, leaving the application halted.

List of qualifiers

The list of qualifiers is dependent on the processor and vehicle and so the GUI does not present things that do not make sense. The command handler generates an error if a specific combination is invalid for a specific processor or vehicle, but this is determined when you issue the command.

To set up a conditional breakpoint, use the following condition qualifiers:

- `macro` (or `;`*`macro-call`*)
- `obj`
- `passcount`
- `when`
- `when_not`.

To specify actions to be performed when a breakpoint triggers, use the following action qualifiers:

- `continue`
- `message`

- `sample` (not supported in this release)
- `timed`
- `update`.

The possible qualifiers are:

`append:(n)`

Instead of creating a new breakpoint, append the qualifiers specified with this command to an existing breakpoint with breakpoint ID number *n*.

Note

You cannot use `append` to change the breakpoint address or to create chained breakpoints.

`continue`

Execution continues when the breakpoint is triggered and no breakpoint details are displayed. Any specified action qualifiers are still performed, depending on the results of any condition qualifiers.

`context:{context}`

Sets the context for other expressions in this breakpoint command to the value of context. This provides an alternative to specifying the complete context for every symbol. For example:

```
BREAKEXECUTION,context:{DHRY_1},when:{Int_1_Loc>0} #152
```

This causes a breakpoint to be set at line 15 of `dhry_1.c` that is triggered only when the variable `status` defined in `dhry_1.c` is greater than zero. The alternative form is:

```
BREAKEXECUTION,when:{Int_1_Loc>0} \DHRY_1\#152
```

Note

When using this alternative form of the command, the filename `DHRY_1` must be uppercase.

`data_only`

The breakpoint is triggered if a data value, specified using `hw_dvalue`, is detected by the debug hardware on the processor data bus.

`gui`

If an error occurs when executing the command or when the breakpoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

Note

This qualifier has no effect in the headless debugger.

`hw_ahigh:(`*`n`*`)`	Specifies the high address for an address-range breakpoint. The low address is specified by the standard breakpoint address. This facility is not supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1000-0x1200`: `BREAKEXECUTION,hw_ahigh:0x1200 0x1000`
`hw_amask:(`*`n`*`)`	Specifies the address mask value for an address-range breakpoint. Addresses that match the standard breakpoint address when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1FA00-0x1FA0F`: `BREAKEXECUTION,hw_amask:0xFFFF0 0x1FA00`
`hw_dvalue:(`*`n`*`)`	Specifies a data value to be compared to values transmitted on the processor data bus. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for the data value `0x400`: `BREAKEXECUTION,hw_dvalue:0x400 0x1FA00`
`hw_dhigh:(`*`n`*`)`	Specifies the high data value for a data-range breakpoint. The low data value is specified by the `hw_dvalue` qualifier. This facility is not supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x00-0x18`: `BREAKEXECUTION,hw_dvalue:0x0,hw_dhigh:0x18 0x1000`
`hw_dmask:(`*`n`*`)`	Specifies the data value mask value for a data-range breakpoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x400-0x4FF`: `BREAKEXECUTION,hw_dvalue:0x400,hw_dmask:0xF00 0x1FA00`

`hw_passcount:(n)`	Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. This qualifier differs from `passcount` only in that it is implemented in hardware. *`n`* is limited to a 32-bit value by the debugger, but might be much more limited by the target hardware, for example to 8 or 16 bits.

You can combine hardware and software pass counts to achieve higher count values. If you define both hardware and software pass counts:

1. when the hardware count reaches zero, the software count is decremented
2. when the software count reaches zero, the breakpoint triggers.

`hw_and:{[then-]id}` Perform an *and* or an *and-then* conjunction with an existing breakpoint. For example, `hw_and:{2}`, or `hw_and:{then-2}`, where *2* is the breakpoint id of another breakpoint.

In the *and* form, the conditions associated with both breakpoints are chained together, so that the action associated with the second breakpoint are performed only when both conditions simultaneously match.

In the *and-then* form, when the condition for the first breakpoint is met, the second breakpoint is enabled but the program is not yet stopped. When the second breakpoint condition is matched, the actions associated are performed. At this point, unless the `continue` qualifier is specified in the second breakpoint, the program stops.

The *`id`* is one of:

- the breakpoint list index of an existing breakpoint
- `next` for the next breakpoint specified for this connection
- `prev` for the last breakpoint specified for this connection.

When a breakpoint is set with `hw_and:next`, and another is set with `hw_and:prev`, these two breakpoints are chained. For example:

```
BREAKEXECUTION,hw_and:next,hw_dvalue:1
    @copyfns\\COPYFNS\mycpy\append
BREAKEXECUTION,hw_and:prev @copyfns\\COPYFNS\mycpy\
```

If you clear the breakpoint that has the ID `next`, then both breakpoints are cleared.

If you clear the breakpoint that has the ID `prev`, then only that breakpoint is cleared.

Debugger internal handle numbers are not available to users to identify breakpoints.

`hw_in:{}`
In trigger tests. The string that follows matches hardware-supported input tests, per vehicle and processor, as a list of names or a value.

`hw_out:{s}`
Not supported in this release.

`hw_not:{s}`
Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *s* can be set to:

`addr`	Invert the breakpoint address value.
`data`	Invert the breakpoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to break when a data value does not match a mask, you can write:

```
BREAKEXECUTION,hw_not:data,hw_dmask:0x00FF ...
```

The break commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
BREAKEXECUTION,hw_not:addr 0x10040
```

This means to break at any address other than `0x10040`. This example is probably not useful.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *nand* and *nand-then* conditions.

This facility is not supported by ARM EmbeddedICE macrocells.

`macro:{`*`MacroCall(arg1,arg2)`*`}`

The triggering of the breakpoint results in the specified macro being executed. Any program variables or functions must be in scope at the time the breakpoint request is entered, or the names must be fully qualified. You must include the braces `{` and `}`.

`message:{"$`*`windowid | fileid`*`$`*`message`*`"}`

Triggering of the breakpoint results in *`message`* being output. Prefixing *`message`* with `$`*`windowid | fileid`*`$` enables you to write the message text to a user-defined window or file. See *Window and file numbers* on page 1-5 for details. For example:

```
BREAKEXECUTION,message:{"$100$this is a message"}
```

`modify:(n)`	Instead of creating a new breakpoint, modify the breakpoint with breakpoint ID number *n* by replacing the address expression and the qualifiers of the existing breakpoint to those specified in this command. **Note** You cannot use this qualifier with the `hw_and` qualifier to change a non-chained breakpoint to a chained breakpoint. However, you can modify a chained breakpoint with any other qualifier and also change the address expression.
`obj:(n)`	This condition is True if the argument *n* matches the C++ object pointer, normally called `this`.
<u>`pass`</u>`count:(n)`	Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. If you specify this in the middle of a sequence of break conditions, those specified before the passcount are processed whether or not the count reaches zero. The conditions specified afterwards are run only when the count reaches zero. There is a hardware passcount qualifier available, `hw_passcount`, for debug hardware that supports it. **Note** If a breakpoint uses a `passcount`, the counting is performed on the host, and so program execution stops briefly every time the breakpoint is hit, even when the count has not been reached.
`sample`	Not supported in this release.
`size:n`	Set the size of the breakpoint to either 16 or 32 bits. For example: `BREAKEXECUTION,size:32 0x10040` Use this qualifier if no debug information is available for your image. By default, RealView Debugger sets a 32-bit breakpoint.
`timed`	Triggering records the time, in whatever units the debug hardware chooses, from the last reset of time. The time can be in nano-seconds, micro-seconds, processor cycles, instructions, or units. See the documentation for the hardware interface for more information. The recorded times are displayed in the Resource Viewer window and in the breakpoint information for this breakpoint.

The timed qualifier can be used for simple profiling, or for a measure of specific response times. If you use `timed` and `continue`, the debugger keeps a log of times for each break.

`update:{"`*`name`*`"}` Update the named windows, or all windows, by reading the memory and processor state when the breakpoint triggers. You can use the name `all` to refresh all windows, or a name specified in the title bar of the window.

This qualifier enables you to get an overview of the process state at a particular point, without having to manually restart the process at each break. The update still takes a significant period of time, and so this method is unsuitable as a non-intrusive debugging tool.

`when:{`*`condition`*`}` The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to True.

Note

Using a macro as an argument to `when`, reverses the sense of the return value from the macro.

`when_not:{`*`condition`*`}`

The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to False.

Examples

The following examples show how to use `BREAKEXECUTION`:

```
BREAKEXECUTION \MATH_1\#449:22
```

Set a hardware breakpoint at line 449, column 22 in the file `math.c`.

```
BREAKE,append:(1),continue,update:{all}
```

Given an already set breakpoint at position 1 in the breakpoint list, add a request to update all windows in the code window for this connection and continue execution each time the breakpoint triggers.

```
BE,hw_pass:(5) \MAIN_1\#49
```

Set a hardware breakpoint using a hardware counter to stop at the fifth time that execution reaches line 49 of `main.c`.

```
BE \MAIN_1\MAIN_C\#33 ;CheckStruct()
```

Set a hardware breakpoint that triggers a call to a debugger macro `CheckStruct` each time it reaches line 33 of `main.c`. If `CheckStruct` returns a nonzero value, the debugger continues application execution.

```
BE,when:{check_struct()} \MAIN_1\#33
```

Set a hardware breakpoint that triggers a call to a target program function `check_struct()` each time it reaches line 33 of `main.c`. If this function returns a zero value, the debugger continues application execution.

Alias

`BEXECUTION` is an alias of `BREAKEXECUTION`.

See also

The following commands provide similar or related functionality:

- *BREAKACCESS* on page 2-43
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *ENABLEBREAK* on page 2-150
- *VMACRO* on page 2-341.

2.3.16 BREAKINSTRUCTION

The `BREAKINSTRUCTION` command sets a software instruction breakpoint at the specified memory location. Software breakpoints are implemented by writing a special instruction at the break address, and so cannot be set in ROM.

See the chapter that describes working with breakpoints in the *RealView Developer Kit v2.2 Debugger User Guide* for full details on the use of breakpoints in RealView Debugger.

Syntax

```
BREAKINSTRUCTION [,qualifier...] expression [=threads,...] [;macro-call]
```

where:

qualifier	Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers* on page 2-62.
expression	Specifies the address at which the breakpoint is placed. For an unqualified breakpoint, this is the address at which program execution is stopped.
threads	The list of threads that make up the break trigger group. Only available for OS-aware RSD connections.
macro-call	Specifies a macro and any parameters it requires. The macro runs when the breakpoint is triggered and before the instruction at the breakpoint is executed. The macro is treated as being specified last in the qualifier list. If the macro returns a nonzero value, or you specified `continue` in the qualifiers, execution continues. If the macro returns a zero value, or if you do not specify a macro, target execution stops and the debugger waits in command mode. The macro argument symbols are interpreted when the breakpoint is specified and so they must be in scope at that point, or you must explicitly qualify them.

Description

`BREAKINSTRUCTION` is used to set or modify software address breakpoints. Address breakpoints include breakpoints set by patching special instructions into the program and hardware that tests the address and data values. If the command has no arguments, it behaves like `DTBREAK` on page 2-135, listing the current breakpoints.

If you try to set a software breakpoint at a location in ROM, and the debugger detects that the attempt failed, and hardware breakpoint facilities are available, the software breakpoint request is retried as a hardware breakpoint request.

You can use qualifiers evaluated in the debugger, such as expressions, macros, C++ object tests, and software pass counters. You can also define actions to occur when the breakpoint is *triggered* (hit), including updating counters or windows, and the enabling or disabling of other breakpoints (see *List of qualifiers*).

When a software breakpoint instruction is hit on the target, the following sequence of events occurs:

1. The debugger associates the address with a specific breakpoint ID. A memory address can only be associated with one user breakpoint at a time.
2. If the breakpoint has a pass count associated with it, the count is updated.
3. The conditions for this breakpoint, if any, are tested in the order specified on the command line. If any condition is False, target execution resumes with the instruction at the breakpointed location. Macros specified with the `macro:` qualifier or the `;`*`macro_call`* argument are run in this phase.
4. If the breakpoint has actions associated with it (for example, using `timed` to note the time the breakpoint occurred) these actions are run, in the order specified on the command line.
5. If the qualifiers include `continue`, target execution resumes with the instruction at the breakpointed location. If not, the debugger updates the state of the GUI and waits for a command, leaving the application halted.

List of qualifiers

The list of qualifiers is dependent on the processor and vehicle and so the GUI does not present things that do not make sense. The command handler generates an error if a specific combination is invalid for a specific processor or vehicle, but this is determined when you issue the command.

To set up a conditional breakpoint, use the following condition qualifiers:

- `macro` (or `;`*`macro-call`*)
- `obj`
- `passcount`
- `when`
- `when_not`.

To specify actions to be performed when a breakpoint triggers, use the following action qualifiers:

- `continue`
- `message`
- `sample` (not supported in this release)
- `timed`
- `update`.

The possible qualifiers are:

`append:(`*`n`*`)`	Instead of creating a new breakpoint, append the qualifiers specified with this command to an existing breakpoint with breakpoint ID number *`n`*. You cannot change the breakpoint address.
<u>`cont`</u>`inue`	Execution continues when the breakpoint is triggered and no breakpoint details are displayed. Any specified action qualifiers are still performed, depending on the results of any condition qualifiers.
`context:{`*`context`*`}`	Sets the context for other expressions in this breakpoint command to the value of context. This provides an alternative to specifying the complete context for every symbol. For example: `bi,context:{DHRY_1},when:{Int_1_Loc>0} #152` This causes a breakpoint to be set at line 15 of `dhry_1.c` that is triggered only when the variable `status` defined in `dhry_1.c` is greater than zero. The alternative form is: `BI,when:{Int_1_Loc>0} \DHRY_1\#152` **Note** When using this alternative form of the command, the filename `DHRY_1` must be uppercase.
`gui`	If an error occurs when executing the command or when the breakpoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane. **Note** This qualifier has no effect in the headless debugger.

`macro:{`*`MacroCall(arg1,arg2)`*`}`

The triggering of the breakpoint results in the specified macro being executed. Any program variables or functions must be in scope at the time the breakpoint request is entered, or the names must be fully qualified. A macro call specified here is treated in the same way as a macro specified after a `;`. You must include the braces `{` and `}`.

`message:{"$`*`windowid | fileid`*`$`*`message`*`"}`

Triggering of the breakpoint results in *`message`* being output. Prefixing *`message`* with `$`*`windowid | fileid`*`$` enables you to write the message text to a user-defined window or file. See *Window and file numbers* on page 1-5 for details. For example:

```
bi,message:{"$100$this is a message"}
```

`modify:(`*`n`*`)` Instead of creating a new breakpoint, modify the breakpoint with breakpoint ID number *`n`* by replacing the address expression and the qualifiers of the existing breakpoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained breakpoint to a chained breakpoint. However, you can modify a chained breakpoint with any other qualifier and also change the address expression.

`obj:(`*`n`*`)` This condition is True if the argument *`n`* matches the C++ object pointer, normally called `this`.

`passcount:(`*`n`*`)` Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. If you specify this in the middle of a sequence of break conditions, those specified before the passcount are processed whether or not the count reaches zero. The conditions specified afterwards are run only when the count reaches zero.

There is a hardware passcount qualifier available, `hw_passcount`, for debug hardware that supports it.

Note

If a hardware breakpoint uses a `passcount`, the counting is performed on the host, and so program execution stops briefly every time the breakpoint is hit, even when the count has not been reached.

`rtos:`*`type`* Sets a breakpoint for OS aware connections, where *`type`* is one of:

`hsd`	Sets a HSD breakpoint for debugging your RTOS image.
`process`	Not supported in this release.
`system`	Sets a system breakpoint for debugging images running in RSD mode.
`thread`	Sets a thread breakpoint for debugging images running in RSD mode.

For more details about these types of breakpoint, see the chapter that describes RTOS support in RealView Developer Kit v2.2 Extensions User Guide.

`sample` Not supported in this release.

`size:`*`n`* Set the size of the breakpoint to either 16 or 32 bits. For example:

```
BREAKINSTRUCTION,size:32 0x10040
```

Use this qualifier if no debug information is available for your image. By default, RealView Debugger sets a 32-bit breakpoint.

`timed` Triggering records the time, in whatever units the debug hardware chooses, from the last reset of time. The time can be in nano-seconds, micro-seconds, processor cycles, instructions, or units. See the documentation for the hardware interface for more information.

The recorded times are displayed in the Resource Viewer window and in the breakpoint information for this breakpoint.

The timed qualifier can be used for simple profiling, or for a measure of specific response times. If you use `timed` and `continue`, the debugger keeps a log of times for each break.

`update:{"`*`name`*`"}` Update the named windows, or all windows, by reading the memory and processor state when the breakpoint triggers. You can use the name `all` to refresh all windows, or a name specified in the title bar of the window.

This qualifier enables you to get an overview of the process state at a particular point, without having to manually restart the process at each break. The update still takes a significant period of time, and so this method is unsuitable as a non-intrusive debugging tool.

`when:{`*`condition`*`}` The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to True.

Note

Using a macro as an argument to `when`, reverses the sense of the return value from the macro.

`when_not:{`*`condition`*`}`

The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to False.

Rules

The following rules apply to the use of the `BREAKINSTRUCTION` command:

- Breakpoints are specific to the board, process, or task active in the window at the time they are set.
- If synchronous breakpoints are set on two or more threads on the same board, the debugger stops the threads as close to the same time as the architecture of the board permits.

Examples

The following examples show how to use `BREAKINSTRUCTION`:

```
BREAKINSTRUCTION \MATH_1\MATH_C\#449:22
```

Set a breakpoint at line 449, column 22 in the file `math.c`.

```
BREAKI,append:(1),continue,update:{all}
```

Given an already set breakpoint at position 1 in the breakpoint list, add a request to update all windows in the code window for this connection and continue execution each time the breakpoint triggers.

```
BI,pass:(5) \MAIN_1\MAIN_C\#49
```

Set a breakpoint using a hardware counter to stop at the fifth time that execution reaches line 49 of `main.c`.

`BI \MAIN_1\MAIN_C\#33 ;CheckStruct()`

Set a breakpoint that triggers a call to a debugger macro `CheckStruct` each time it reaches line 33 of `main.c`. If `CheckStruct` returns a nonzero value, the debugger continues application execution.

`BI,when:{count<4 || err==5} \MAIN_1\SUBFN_C\#33`

Set a breakpoint that triggers when the expression `count<4 || err==5` is True when execution reaches line 33 of `subfn.c`.

`BI,when:{check_struct()} \MAIN_1\MAIN_C\#33`

Set a breakpoint that triggers a call to a target program function `check_struct()` each time it reaches line 33 of `main.c`. If this function returns a zero value, the debugger continues application execution.

`breakinstruction, rtos:hsd \DEMO\#201`

Set a HSD breakpoint at line `201` in `demo.c`.

`bi,rtos:system \DEMO\#154`

Set a system breakpoint at line `154` in `demo.c`.

`bi,rtos:thread \DEMO\#154 = 0x39d8, 0x3a68`

Set a thread breakpoint using a break trigger group consisting of two threads, defined by the addresses of the thread control blocks.

`bi,rtos:thread \DEMO\#180 = thread_2, thread_6, thread_8`

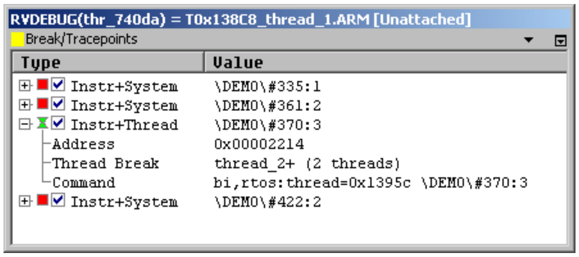
Set a thread breakpoint using a break trigger group consisting of three threads, defined by the thread names.

`bi,modify:2,rtos:system`

Modify breakpoint number 2, a thread breakpoint, to be a system breakpoint.

`bi,modify:3,rtos:thread = 0x1395c, 0x13bac`

Modify breakpoint number 3, a thread breakpoint, to specify a different break trigger group, shown in Figure 2-1 on page 2-68.



```
RVDEBUG(thr_740da) = T0x138C8_thread_1.ARM [Unattached]
Break/Tracepoints
Type                 Value
⊞ ■ ☑ Instr+System   \DEMO\#335:1
⊞ ■ ☑ Instr+System   \DEMO\#361:2
⊟ ⧗ ☑ Instr+Thread   \DEMO\#370:3
    ├Address         0x00002214
    ├Thread Break    thread_2+ (2 threads)
    └Command         bi,rtos:thread=0x1395c \DEMO\#370:3
⊞ ■ ☑ Instr+System   \DEMO\#422:2
```

Figure 2-1 Changing the break trigger group

Alias

BINSTRUCTION and BREAK are aliases of BREAKINSTRUCTION.

See also

The following commands provide similar or related functionality:

- *AOS_resource-list* on page 2-32
- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *CLEARBREAK* on page 2-94
- *DOS_resource-list* on page 2-129
- *ENABLEBREAK* on page 2-150
- *OSCTRL* on page 2-214
- *STOP* on page 2-283
- *VMACRO* on page 2-341.

2.3.17 BREAKREAD

The `BREAKREAD` command sets a hardware breakpoint that triggers when a read operation is performed on any of the specified memory locations.

See the chapter that describes working with breakpoints in the *RealView Developer Kit v2.2 Debugger User Guide* for full details on the use of breakpoints in RealView Debugger.

Syntax

```
BREAKREAD [,qualifier...] {address | address_range} [;macro_call]
```

where:

qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers* on page 2-62.

address | address_range

Specifies a single address in target memory, or an address range. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

macro_call Specifies a macro and any parameters it requires. The macro runs when the breakpoint is triggered and before the instruction at the breakpoint is executed. The macro is treated as being specified last in the qualifier list.

If the macro returns a nonzero value, or you specified `continue` in the qualifiers, execution continues. If the macro returns a zero value, or if you do not specify a macro, target execution stops and the debugger waits in command mode.

The macro argument symbols are interpreted when the breakpoint is specified and so they must be in scope at that point, or you must explicitly qualify them.

Description

`BREAKREAD` is used to set or modify data read breakpoints. Data read breakpoints trigger when data that matches a condition is read from memory at a particular address or address range. If the command has no arguments, it behaves like `DTBREAK` on page 2-135, listing the current breakpoints.

You can use qualifiers evaluated in the debugger, such as expressions, macros, C++ object tests, and software pass counters. You can also define actions to occur when the breakpoint is *triggered* (hit), including updating counters or windows, and the enabling or disabling of other breakpoints (see *List of qualifiers* on page 2-62).

If you do not specify an address, the read breakpoint is set at the address defined by the current value of the PC. The breakpoint is triggered if the target program reads data from any specified target memory area.

When a hardware data read breakpoint is hit on the target, the following sequence of events occurs:

1. The debugger or the hardware associates the event with a specific debugger breakpoint ID.
2. If the breakpoint has a software pass count associated with it, the count is updated.
3. The conditions for this breakpoint, if any, are tested in the order specified on the command line. If any condition is False, target execution resumes with the instruction at the breakpointed location. Macros specified with the `macro:` qualifier or the `;`*`macro_call`* argument are run in this phase.
4. If the breakpoint has actions associated with it (for example, using `timed` to note the time the breakpoint occurred) these actions are run, in the order specified on the command line.
5. If the qualifiers include `continue`, target execution resumes with the instruction at the breakpointed location. If not, the debugger updates the state of the GUI and waits for a command, leaving the application halted.

List of qualifiers

The list of qualifiers is dependent on the processor and vehicle and so the GUI does not present things that do not make sense. The command handler generates an error if a specific combination is invalid for a specific processor or vehicle, but this is determined when you issue the command.

To set up a conditional breakpoint, use the following condition qualifiers:

- `macro` (or `;`*`macro-call`*)
- `obj`
- `passcount`
- `when`
- `when_not`.

To specify actions to be performed when a breakpoint triggers, use the following action qualifiers:

- `continue`
- `message`
- `sample` (not supported in this release)

- `timed`
- `update`.

The possible qualifiers are:

`append:(`*n*`)` Instead of creating a new breakpoint, append the qualifiers specified with this command to an existing breakpoint with breakpoint ID number *n*.

——— **Note** ———

You cannot use `append` to change the breakpoint address or to create chained breakpoints.

———————————

`continue` Execution continues when the breakpoint is triggered and no breakpoint details are displayed. Any specified action qualifiers are still performed, depending on the results of any condition qualifiers.

`context:{`*context*`}` Sets the context for other expressions in this breakpoint command to the value of context. This provides an alternative to specifying the complete context for every symbol. For example:

```
BREAKREAD,context:{DHRY_1},when:{Int_1_Loc>0} #152
```

This causes a breakpoint to be set at line 15 of `hello.c` that is triggered only when the variable `status` defined in `hello.c` is greater than zero. The alternative form is:

```
BREAKREAD,when:{Int_1_Loc>0} \DHRY_1\#152
```

——— **Note** ———

When using this alternative form of the command, the filename `DHRY_1` must be uppercase.

———————————

`data_only` The breakpoint is triggered if a data value, specified using `hw_dvalue`, is detected by the debug hardware on the processor data bus.

`gui` If an error occurs when executing the command or when the breakpoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

——— **Note** ———

This qualifier has no effect in the headless debugger.

———————————

`hw_ahigh:(`*n*`)`	Specifies the high address for an address-range breakpoint. The low address is specified by the standard breakpoint address. This facility is not supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1000-0x1200`: `BREAKREAD,hw_ahigh:0x1200 0x1000`
`hw_amask:(`*n*`)`	Specifies the address mask value for an address-range breakpoint. Addresses that match the standard breakpoint address when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1FA00-0x1FA0F`: `BREAKREAD,hw_amask:0xFFFF0 0x1FA00`
`hw_dvalue:(`*n*`)`	Specifies a data value to be compared to values transmitted on the processor data bus. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for the data value `0x400`: `BREAKREAD,hw_dvalue:0x400 0x1FA00`
`hw_dhigh:(`*n*`)`	Specifies the high data value for a data-range breakpoint. The low data value is specified by the `hw_dvalue` qualifier. This facility is not supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x00-0x18`: `BREAKREAD,hw_dvalue:0x0,hw_dhigh:0x18 0x1000`
`hw_dmask:(`*n*`)`	Specifies the data value mask value for a data-range breakpoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x400-0x4FF`: `BREAKREAD,hw_dvalue:0x400,hw_dmask:0xF00 0x1FA00`

`hw_passcount:(`*`n`*`)`

Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. This qualifier differs from `passcount` only in that it is implemented in hardware. *`n`* is limited to a 32-bit value by the debugger, but might be much more limited by the target hardware, for example to 8 or 16 bits.

You can combine hardware and software pass counts to achieve higher count values. If you define both hardware and software pass counts:

1. when the hardware count reaches zero, the software count is decremented
2. when the software count reaches zero, the breakpoint triggers.

`hw_and:{[then-]`*`id`*`}`

Perform an *and* or an *and-then* conjunction with an existing breakpoint. For example, `hw_and:{2}`, or `hw_and:{then-2}`, where *2* is the breakpoint id of another breakpoint.

In the *and* form, the conditions associated with both breakpoints are chained together, so that the action associated with the second breakpoint are performed only when both conditions simultaneously match.

In the *and-then* form, when the condition for the first breakpoint is met, the second breakpoint is enabled but the program is not yet stopped. When the second breakpoint condition is matched, the actions associated are performed. At this point, unless the `continue` qualifier is specified in the second breakpoint, the program stops.

The *`id`* is one of:

- the breakpoint list index of an existing breakpoint
- `next` for the next breakpoint specified for this connection
- `prev` for the last breakpoint specified for this connection.

When a breakpoint is set with `hw_and:next`, and another is set with `hw_and:prev`, these two breakpoints are chained. For example:

```
BREAKREAD,hw_and:next,hw_dvalue:1
    @copyfns\\COPYFNS\mycpy\append
BREAKREAD,hw_and:prev @copyfns\\COPYFNS\mycpy\
```

If you clear the breakpoint that has the ID `next`, then both breakpoints are cleared.

If you clear the breakpoint that has the ID `prev`, then only that breakpoint is cleared.

Debugger internal handle numbers are not available to users to identify breakpoints.

`hw_in:{}` In trigger tests. The string that follows matches hardware-supported input tests, per vehicle and processor, as a list of names or a value.

`hw_out:{s}` Not supported in this release.

`hw_not:{s}` Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *`s`* can be set to:

`addr`	Invert the breakpoint address value.
`data`	Invert the breakpoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to break when a data value does not match a mask, you can write:

```
BREAKREAD,hw_not:data,hw_dmask:0x00FF ...
```

The break commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
BREAKREAD,hw_not:addr 0x10040
```

This means to break at any address other than `0x10040`. This example is probably not useful.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *nand* and *nand-then* conditions.

This facility is not supported by ARM EmbeddedICE macrocells.

`macro:{`*`MacroCall(arg1,arg2)`*`}`

The triggering of the breakpoint results in the specified macro being executed. Any program variables or functions must be in scope at the time the breakpoint request is entered, or the names must be fully qualified. You must include the braces `{` and `}`.

<u>`mess`</u>`age:{"$`*`windowid`*` | `*`fileid`*`$`*`message`*`"}`

Triggering of the breakpoint results in *`message`* being output. Prefixing *`message`* with `$`*`windowid`*` | `*`fileid`*`$` enables you to write the message text to a user-defined window or file. See *Window and file numbers* on page 1-5 for details. For example:

```
breakread,message:{"$100$this is a message"}
```

`modify:(n)`
: Instead of creating a new breakpoint, modify the breakpoint with breakpoint ID number *n* by replacing the address expression and the qualifiers of the existing breakpoint to those specified in this command.

  Note

  You cannot use this qualifier with the `hw_and` qualifier to change a non-chained breakpoint to a chained breakpoint. However, you can modify a chained breakpoint with any other qualifier and also change the address expression.

`obj:(n)`
: This condition is True if the argument *n* matches the C++ object pointer, normally called `this`.

`passcount:(n)`
: Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. If you specify this in the middle of a sequence of break conditions, those specified before the passcount are processed whether or not the count reaches zero. The conditions specified afterwards are run only when the count reaches zero.

  There is a hardware passcount qualifier available, `hw_passcount`, for debug hardware that supports it.

  Note

  If a hardware breakpoint uses a `passcount`, the counting is performed on the host, and so program execution stops briefly every time the breakpoint is hit, even when the count has not been reached.

`sample`
: Not supported in this release.

`size:n`
: Set the size of the breakpoint to either 16 or 32 bits. For example:

  ```
  BREAKREAD,size:32 0x10040
  ```

  Use this qualifier if no debug information is available for your image. By default, RealView Debugger sets a 32-bit breakpoint.

`register:expression`
: The breakpoint is triggered if the value stored in the specified memory-mapped register is read. The register is identified by *expression*. For example:

  ```
  BREAKREAD,register:PR1
  ```

  or

  ```
  BREAKREAD,register:@PR1
  ```

	Note You can only specify registers that are defined in a Board/Chip Definition (`.bcd`) file, which you must have assigned to the connection.
`timed`	Triggering records the time, in whatever units the debug hardware chooses, from the last reset of time. The time can be in nano-seconds, micro-seconds, processor cycles, instructions, or units. See the documentation for the hardware interface for more information. The recorded times are displayed in the Resource Viewer window and in the breakpoint information for this breakpoint. The timed qualifier can be used for simple profiling, or for a measure of specific response times. If you use `timed` and `continue`, the debugger keeps a log of times for each break.
`update:{"`*`name`*`"}`	Update the named windows, or all windows, by reading the memory and processor state when the breakpoint triggers. You can use the name `all` to refresh all windows, or a name specified in the title bar of the window. This qualifier enables you to get an overview of the process state at a particular point, without having to manually restart the process at each break. The update still takes a significant period of time, and so this method is unsuitable as a non-intrusive debugging tool.
`when:{`*`condition`*`}`	The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to True. **Note** Using a macro as an argument to `when`, reverses the sense of the return value from the macro.
`when_not:{`*`condition`*`}`	The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to False.

Examples

The following examples show how to use `BREAKREAD`:

`BREAKREAD 0x8000`	Stop program execution if a read occurs at location 0x8000.

```
BREAKREAD 0x100..0x200
```

Stop program execution if a read occurs in the 257 bytes from 0x100-0x200 (inclusive).

Alias

`BREAD` is an alias of `BREAKREAD`.

See also

The following commands provide similar or related functionality:

- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKWRITE* on page 2-78
- *CLEARBREAK* on page 2-94
- *DTBREAK* on page 2-135
- *ENABLEBREAK* on page 2-150
- *VMACRO* on page 2-341.

2.3.18 BREAKWRITE

The `BREAKWRITE` command sets a hardware breakpoint that triggers when a write operation is performed on any of the specified memory locations.

See the chapter that describes working with breakpoints in the *RealView Developer Kit v2.2 Debugger User Guide* for full details on the use of breakpoints in RealView Debugger.

Syntax

```
BREAKWRITE [,qualifier...] {address | address_range} [; macro_call]
```

where:

qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers* on page 2-79.

address | *address_range*

Specifies a single address in target memory, or an address range. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

macro_call Specifies a macro and any parameters it requires. The macro runs when the breakpoint is triggered and before the instruction at the breakpoint is executed. The macro is treated as being specified last in the qualifier list.

If the macro returns a nonzero value, or you specified `continue` in the qualifiers, execution continues. If the macro returns a zero value, or if you do not specify a macro, target execution stops and the debugger waits in command mode.

The macro argument symbols are interpreted when the breakpoint is specified and so they must be in scope at that point, or you must explicitly qualify them.

Description

`BREAKWRITE` is used to set or modify data write breakpoints. Data write breakpoints trigger when data that matches a condition is written to memory at a particular address or address range. If the command has no arguments, it behaves like `DTBREAK` on page 2-135, listing the current breakpoints.

You can use qualifiers evaluated in the debugger, such as expressions, macros, C++ object tests, and software pass counters. You can also define actions to occur when the breakpoint is *triggered* (hit), including updating counters or windows, and the enabling or disabling of other breakpoints (see *List of qualifiers* on page 2-79).

If you do not specify an address, the write breakpoint is set at the address defined by the current value of the PC. The breakpoint is triggered if the target program writes data to any part of the specified target memory area.

When a hardware data write breakpoint is hit on the target, the following sequence of events occurs:

1. The debugger or the hardware associates the event with a specific debugger breakpoint ID.
2. If the breakpoint has a software pass count associated with it, the count is updated.
3. The conditions for this breakpoint, if any, are tested in the order specified on the command line. If any condition is False, target execution resumes with the instruction at the breakpointed location. Macros specified with the `macro:` qualifier or the `;`*`macro_call`* argument are run in this phase.
4. If the breakpoint has actions associated with it (for example, using `timed` to note the time the breakpoint occurred) these actions are run, in the order specified on the command line.
5. If the qualifiers include `continue`, target execution resumes with the instruction at the breakpointed location. If not, the debugger updates the state of the GUI and waits for a command, leaving the application halted.

List of qualifiers

The list of qualifiers is dependent on the processor and vehicle and so the GUI does not present things that do not make sense. The command handler generates an error if a specific combination is invalid for a specific processor or vehicle, but this is determined when you issue the command.

To set up a conditional breakpoint, use the following condition qualifiers:

- `macro` (or `;`*`macro-call`*)
- `obj`
- `passcount`
- `when`
- `when_not`.

To specify actions to be performed when a breakpoint triggers, use the following action qualifiers:

- `continue`
- `message`
- `sample` (not supported in this release)

- `timed`
- `update`.

The possible qualifiers are:

`append:(`*n*`)`
: Instead of creating a new breakpoint, append the qualifiers specified with this command to an existing breakpoint with breakpoint ID number *n*.

Note

You cannot use `append` to change the breakpoint address or to create chained breakpoints.

`continue`
: Execution continues when the breakpoint is triggered and no breakpoint details are displayed. Any specified action qualifiers are still performed, depending on the results of any condition qualifiers.

`context:{`*context*`}`
: Sets the context for other expressions in this breakpoint command to the value of context. This provides an alternative to specifying the complete context for every symbol. For example:

```
BREAKWRITE,context:{DHRY_1},when:{Int_1_Loc>0} #152
```

This causes a breakpoint to be set at line 15 of `hello.c` that is triggered only when the variable `status` defined in `hello.c` is greater than zero. The alternative form is:

```
BREAKWRITE,when:{Int_1_Loc>0} \DHRY_1\#152
```

Note

When using this alternative form of the command, the filename `DHRY_1` must be uppercase.

`data_only`
: The breakpoint is triggered if a data value, specified using `hw_dvalue`, is detected by the debug hardware on the processor data bus.

`gui`
: If an error occurs when executing the command or when the breakpoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

Note

This qualifier has no effect in the headless debugger.

`hw_ahigh:(n)`	Specifies the high address for an address-range breakpoint. The low address is specified by the standard breakpoint address. This facility is not supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1000-0x1200`: `BREAKWRITE,hw_ahigh:0x1200 0x1000`
`hw_amask:(n)`	Specifies the address mask value for an address-range breakpoint. Addresses that match the standard breakpoint address when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any address between `0x1FA00-0x1FA0F`: `BREAKWRITE,hw_amask:0xFFFF0 0x1FA00`
`hw_dvalue:(n)`	Specifies a data value to be compared to values transmitted on the processor data bus. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for the data value `0x400`: `BREAKWRITE,hw_dvalue:0x400 0x1FA00`
`hw_dhigh:(n)`	Specifies the high data value for a data-range breakpoint. The low data value is specified by the `hw_dvalue` qualifier. This facility is not supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x00-0x18`: `BREAKWRITE,hw_dvalue:0x0,hw_dhigh:0x18 0x1000`
`hw_dmask:(n)`	Specifies the data value mask value for a data-range breakpoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the breakpoint to trigger. This facility is supported by ARM EmbeddedICE macrocells. For example, this command sets a breakpoint that triggers for any data value between `0x400-0x4FF`: `BREAKWRITE,hw_dvalue:0x400,hw_dmask:0xF00 0x1FA00`

`hw_passcount:(n)`
: Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. This qualifier differs from `passcount` only in that it is implemented in hardware. *n* is limited to a 32-bit value by the debugger, but might be much more limited by the target hardware, for example to 8 or 16 bits.

You can combine hardware and software pass counts to achieve higher count values. If you define both hardware and software pass counts:

1. when the hardware count reaches zero, the software count is decremented
2. when the software count reaches zero, the breakpoint triggers.

`hw_and:{[then-]id}`
: Perform an *and* or an *and-then* conjunction with an existing breakpoint. For example, `hw_and:{2}`, or `hw_and:{then-2}`, where *2* is the breakpoint id of another breakpoint.

In the *and* form, the conditions associated with both breakpoints are chained together, so that the action associated with the second breakpoint are performed only when both conditions simultaneously match.

In the *and-then* form, when the condition for the first breakpoint is met, the second breakpoint is enabled but the program is not yet stopped. When the second breakpoint condition is matched, the actions associated are performed. At this point, unless the `continue` qualifier is specified in the second breakpoint, the program stops.

The *id* is one of:

- the breakpoint list index of an existing breakpoint
- `next` for the next breakpoint specified for this connection
- `prev` for the last breakpoint specified for this connection.

When a breakpoint is set with `hw_and:next`, and another is set with `hw_and:prev`, these two breakpoints are chained. For example:

```
BREAKWRITE,hw_and:next,hw_dvalue:1
    @copyfns\\COPYFNS\mycpy\append
BREAKWRITE,hw_and:prev @copyfns\\COPYFNS\mycpy\
```

If you clear the breakpoint that has the ID `next`, then both breakpoints are cleared.

If you clear the breakpoint that has the ID `prev`, then only that breakpoint is cleared.

Debugger internal handle numbers are not available to users to identify breakpoints.

`hw_in:{}`	In trigger tests. The string that follows matches hardware-supported input tests, per vehicle and processor, as a list of names or a value.
`hw_out:{`*`s`*`}`	Not supported in this release.

`hw_not:{`*`s`*`}`

Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *`s`* can be set to:

`addr`	Invert the breakpoint address value.
`data`	Invert the breakpoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to break when a data value does not match a mask, you can write:

```
BREAKWRITE,hw_not:data,hw_dmask:0x00FF ...
```

The break commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
BREAKWRITE,hw_not:addr 0x10040
```

This means to break at any address other than `0x10040`. This example is probably not useful.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *nand* and *nand-then* conditions.

This facility is not supported by ARM EmbeddedICE macrocells.

`macro:{`*`MacroCall(arg1,arg2)`*`}`

The triggering of the breakpoint results in the specified macro being executed. Any program variables or functions must be in scope at the time the breakpoint request is entered, or the names must be fully qualified. You must include the braces `{` and `}`.

`message:{"$`*`windowid | fileid`*`$`*`message`*`"}`

Triggering of the breakpoint results in *`message`* being output. Prefixing *`message`* with `$`*`windowid | fileid`*`$` enables you to write the message text to a user-defined window or file. See *Window and file numbers* on page 1-5 for details. For example:

```
BREAKWRITE,message:{"$100$this is a message"}
```

`modify:(n)`	Instead of creating a new breakpoint, modify the breakpoint with breakpoint ID number *n* by replacing the address expression and the qualifiers of the existing breakpoint to those specified in this command. **Note** You cannot use this qualifier with the `hw_and` qualifier to change a non-chained breakpoint to a chained breakpoint. However, you can modify a chained breakpoint with any other qualifier and also change the address expression.
`obj:(n)`	This condition is True if the argument *n* matches the C++ object pointer, normally called `this`.
`passcount:(n)`	Specifies the number of times that the break condition is ignored before it is triggered. The default value is 0. If you specify this in the middle of a sequence of break conditions, those specified before the passcount are processed whether or not the count reaches zero. The conditions specified afterwards are run only when the count reaches zero. There is a hardware passcount qualifier available, `hw_passcount`, for debug hardware that supports it. **Note** If a hardware breakpoint uses a `passcount`, the counting is performed on the host, and so program execution stops briefly every time the breakpoint is hit, even when the count has not been reached.

`register:expression`	The breakpoint is triggered if a value is written to the specified memory-mapped register. The register is identified by *expression*. For example:
	`BREAKWRITE,register:PR1`
	or
	`BREAKWRITE,register:@PR1`
	Note
	You can only specify registers that are defined in a Board/Chip Definition (`.bcd`) file, which you must have assigned to the connection.

`sample`	Not supported in this release.
`size:`*`n`*	Set the size of the breakpoint to either 16 or 32 bits. For example: `BREAKWRITE,size:32 0x10040` Use this qualifier if no debug information is available for your image. By default, RealView Debugger sets a 32-bit breakpoint.
`timed`	Triggering records the time, in whatever units the debug hardware chooses, from the last reset of time. The time can be in nano-seconds, micro-seconds, processor cycles, instructions, or units. See the documentation for the hardware interface for more information. The recorded times are displayed in the Resource Viewer window and in the breakpoint information for this breakpoint. The timed qualifier can be used for simple profiling, or for a measure of specific response times. If you use `timed` and `continue`, the debugger keeps a log of times for each break.
`update:{"`*`name`*`"}`	Update the named windows, or all windows, by reading the memory and processor state when the breakpoint triggers. You can use the name `all` to refresh all windows, or a name specified in the title bar of the window. This qualifier enables you to get an overview of the process state at a particular point, without having to manually restart the process at each break. The update still takes a significant period of time, and so this method is unsuitable as a non-intrusive debugging tool.
`when:{`*`condition`*`}`	The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to True. ——— **Note** ——— Using a macro as an argument to `when`, reverses the sense of the return value from the macro. ———————————
`when_not:{`*`condition`*`}`	The breakpoint is triggered whenever *condition*, a debugger expression, evaluates to False.

Examples

The following examples show how to use `BREAKWRITE`:

`BREAKWRITE 0x8000`	Stop program execution if the program writes to location 0x8000.
`BREAKW 0x1100..0x1200`	Stop program execution if the program writes to the 257 bytes from `0x1100-0x1200` (inclusive).
`BWRITE 0x1100..0x1200 ; CheckMem(0x100)`	Stop program execution if the program writes to the 257 bytes from `0x1100-0x1200` (inclusive) and calls the macro `CheckMem` with the base address `0x100`.

Alias

`BWRITE` is an alias of `BREAKWRITE`.

See also

The following commands provide similar or related functionality:

- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *CLEARBREAK* on page 2-94
- *DTBREAK* on page 2-135
- *ENABLEBREAK* on page 2-150
- *VMACRO* on page 2-341.

2.3.19 BROWSE

The BROWSE command invokes the C++ class browser interface

Syntax

<u>BROW</u>SE *symbol*

where:

symbol	Specifies a C++ class or structure to be browsed.

Description

Displays the parent class or classes and any child classes for the class you specify. You can specify the class as either a variable name or the class name.

Examples

The following example shows how to use BROWSE:

```
browse Shakespeare
                   Shakespeare
  parents             |                     children
   ____________________|____________________
  /                                         \
  | baseclass
```

See also

There are no other commands that provide similar or related functionality.

2.3.20 BWRITE

`BWRITE` is an alias of `BREAKWRITE` (see page 2-78).

2.3.21 CANCEL

The `CANCEL` command cancels, or interrupts, the execution of commands.

Syntax

```
CANCEL
```

Description

The `CANCEL` command enables you to interrupt, or cancel, an asynchronous command that the is still executing. It is equivalent to the Cancel toolbar icon. If the target is running, only commands that can definitely be run with a running target are executed. Other commands are held in a queue for execution when the target stops. This is called *pending* the command. Use the `CANCEL` command to clear pending commands from the list, to stop them being executed.

You cannot use this command to halt target execution. Use `HALT` to do this.

Note

Synchronous commands can only be run when target program execution has stopped.

Asynchronous commands can be run at all times.

See also

The following commands provide similar or related functionality:

- *HALT* on page 2-177
- *INTRPT* on page 2-183
- *WAIT* on page 2-346.

2.3.22 CCTRL

The `CCTRL` command opens and closes the Connection Control window.

——— **Note** ———

This command has no effect in the headless debugger.

————————————

Syntax

```
CCTRL
```

Description

The `CCTRL` command enables you to open and close the Connection Control window. If the Connection Control window is open, then the command closes the window. If the Connection Control window is closed, then the command opens the window.

2.3.23 CEXPRESSION

The CEXPRESSION command calculates and displays the value of an expression. You can also modify variables using the assignment operator.

Syntax

```
CEXPRESSION [/R] expression
```

where:

/R	Suppresses printing of the result, that is the line beginning with the text `Result is:`.
expression	A valid debugger expression.

Description

The CEXPRESSION command calculates the value of an expression or assigns a value to a variable. Debugger expressions are described in more detail in Chapter 1 *Working with the CLI* but include target function and procedure calls, debugger macro invocation, and scalar C languages expressions. You cannot manipulate values larger than 4 bytes, other than **double** values, in an expression.

If you use CEXPRESSION to run a target function, it is called using the target resources, including stack and heap space. The debugger ensures that the core processor registers are saved before calling the debugger function and restored afterwards. The following issues must be remembered when calling target application functions:

- Target function calls must be supported for your processor.
- You must ensure that the target has initialized those resources that the called function, and any function it calls, requires.

  This normally requires at least that the C runtime code has completed execution so that the stack and heap are set up.
- If the target function has side effects, for example changing global variables, the side effects might not be reflected in the original application straight away, because the compiler might have stored elements of that global state in registers, or even indirectly in the PC. It is likely that programs compiled with optimization enabled are more prone to this issue.

Rules

The following rules apply to the use of the `CEXPRESSION` command:

- `CEXPRESSION` runs synchronously if the expression uses target registers, including the stack pointer, or if it uses target memory and background memory access is not available.

  Use the `WAIT` command to force it to run synchronously.
- You must have a valid target execution context before you can run target functions correctly.
- Macros take higher precedence than target functions. If a target function and a macro have the same name, the macro is the one that is executed unless the target function is qualified.
- Results are displayed in either floating-point format, address format, or in decimal, hexadecimal, or ASCII format depending on the type of variables used in the expression.
- The ASCII representation is displayed if the expression value is a printable ASCII character.
- Floating-point numbers are shown as double by default (14 decimal digits of precision). They can be cast to float to display 6 decimal digits of precision.

Examples

The following examples show how to use `CEXPRESSION`:

`CEXPRESSION Run_Index`	
	Displays the current value of the variable named `Run_Index`.
`CE /R Run_Index=50`	Assigns a value of `50` to the variable named `Run_Index`, and suppresses the printing of the result.
`CE sin(0.2)`	Displays the value of the application function `sin()`, passing in the value `(double)0.2`.
`CE @R0 =20h`	Writes `0x20` to target register `R0`. For more details on operations with registers, see *Referencing reserved symbols* on page 1-15.

See also

The following commands provide similar or related functionality:

- *ADD* on page 2-18

- *DEFINE* on page 2-110
- *DUMP* on page 2-141
- *MACRO* on page 2-195
- *PRINTVALUE* on page 2-229
- *SETMEM* on page 2-257
- *SETREG* on page 2-259.

2.3.24 CLEARBREAK

The `CLEARBREAK` command deletes one or more breakpoints.

Syntax

`CLEARBREAK [{breakpoint_number | breakpoint_number_range}]`

where:

breakpoint_number

Specifies the breakpoint number to be cleared.

breakpoint_number_range

Specifies a range of breakpoint numbers as two integers separated by the range operator (`..`).

Description

This command clears (deletes) the breakpoints you specify, using the position of the breakpoint in a list of breakpoints to identify the breakpoints to clear.

You can display a list of the currently defined breakpoints using the command `DTBREAK` (see page 2-135), and also by displaying the Break/Tracepoints pane in the Code window.

When specifying a range of breakpoints, you can either specify the end of the range as an absolute position, or you can specify the number of breakpoints to delete by typing a plus sign followed by the number of breakpoints. For example: `+3` indicates three breakpoints.

To delete all breakpoints, use `CLEARBREAK` with no parameters.

`CLEARBREAK` runs synchronously.

——— **Note** ———

You can disable a breakpoint, so that the breakpoint is unset but remembered by the debugger, using the `DISABLEBREAK` command. You can enable breakpoints that you have disabled, so setting them on the target again, using the `ENABLEBREAK` command.

———————————

Examples

`CL`	Clears every breakpoint.
`CL 5`	Clears the breakpoint listed fifth in the current list of breakpoints.

`CL 5..7`	Clears the fifth, sixth, and seventh breakpoints in the current list.
`CL 5..+3`	Clears the fifth, sixth, and seventh breakpoints in the current list.

See also

The following commands provide similar or related functionality:

- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *DISABLEBREAK* on page 2-120
- *DTBREAK* on page 2-135
- *ENABLEBREAK* on page 2-150.

2.3.25 COMPARE

The `COMPARE` command compares two blocks of memory and displays the differences.

Syntax

`COMPARE [/R] [address_range, address]`

where:

`/R` Instructs the debugger to continue comparing and displaying mismatches until the end of the block is reached or until CTRL-Break is pressed.

`address_range`

Specifies the address range to be compared using two addresses separated by the range operator (`..`). See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

`address` Specifies the starting address of the block of memory to use as a comparison.

Description

A specified block of memory is compared to a block of the same size starting at a specified location.

Mismatched addresses and values are displayed. If you are using the GUI, then they are displayed in the Output pane. Entering the command again at this point without parameters continues the process starting with the first byte after the mismatch.

If the contents of the two blocks of memory are the same, the debugger displays the message:

```
Memory blocks are the same.
```

`COMPARE` runs synchronously unless background access to target memory is supported. Use the `WAIT` command to force it to run synchronously.

Examples

The following examples show how to use `COMPARE`:

```
com 0x8100..0x82FF,0x8700
```

Compares the contents of memory from `0x8100` to `0x82FF` with the contents of memory from `0x8700` to `088FF`, stopping at the first mismatch.

```
com/r 0x8100..0x81FF,0x8700
```

Compares the contents of memory from `0x8100` to `0x81FF` with the contents of memory from `0x8700` to `087FF`, displaying all the differences found.

```
com/r 0x8100..+512,0x8700
```

Compares the contents of memory from `0x8100` to `0x81FF` with the contents of memory from `0x8700` to `088FF`, displaying all the differences found.

See also

The following commands provide similar or related functionality:

- *COPY* on page 2-104
- *FILL* on page 2-161
- *MEMWINDOW* on page 2-201
- *TEST* on page 2-285
- *VERIFYFILE* on page 2-338.

2.3.26 CONNECT

The `CONNECT` command is used to connect the debugger to a specified target.

Syntax

```
CONNECT [,route] [{,reset|,noreset}] [{,halt|,nohalt}] [=] {targetid |
@targetname}
```

```
CONNECT [,gui] [=] {targetid | @targetname}
```

where:

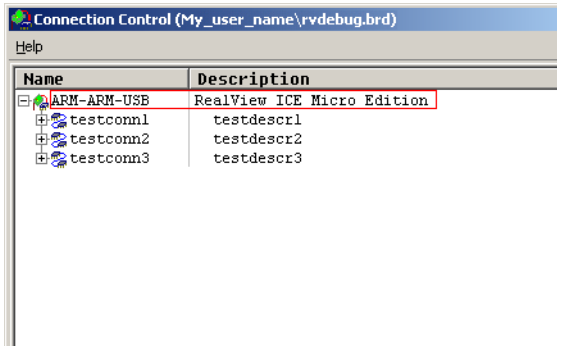
`route`	Indicates that the specified *targetid* is a target access-provider, not the final target device.
`reset`	Reset the target before connecting to it.
`noreset`	Do not reset the target on connecting to it.
`halt`	Stop the target on connecting to it.
`nohalt`	Do not stop the target on connecting to it.
`targetid`	Specifies the required target as a number. See *Making numbered connections* on page 2-100 for details.
`targetname`	Specifies the required target as a name. See *Making named connections* on page 2-101 for details.
`gui`	Enables you to choose the connect mode from a dialog or prompt: • If you use this option in the GUI CLI, it displays a dialog. • If you use this option in the headless debugger, it displays a prompt. The connect specifies what state you want the debugger to leave the target in after the connection. See *Connect modes* on page 2-99 for more details.

Description

The `CONNECT` command creates a new target connection. The details of the connection are specified using the board file. To connect to a target you indicate which target in the board file you want to connect to. There are two ways to specify the target:

- As a number
- As an identifier string.

Using the `CONNECT` command means that you do not use the Connection Control window (shown in Figure 2-2). However, it is helpful to think of that window when considering the operation of the `CONNECT` command.



Name	Description
ARM-ARM-USB	RealView ICE Micro Edition
testconn1	testdescr1
testconn2	testdescr2
testconn3	testdescr3

Figure 2-2 Connection Control window

——— **Note** ———

If you set the connect mode in the board (`.BRD`) file of the target, or the `.BCD` file for a board, the target connects using that mode. If you specify `prompt` for the connect mode, then the `CONNECT` command acts as though you specified the `,gui` qualifier. The `reset`, `noreset`, `halt`, and `nohalt` qualifiers override the connect mode setting in the board file.

————————————

Connect modes

When you connect to a target, the connect mode determines what happens to the target:

No Reset and Stop

Connect to the target, but do not reset it. If the target is running, stop it. This is the default.

No Reset and No Stop

Connect to the target, but do not reset it. The running state of the target is unchanged.

Reset and Stop

Connect to the target, and reset it. If the target is running after the reset, stop it.

Reset and No Stop

Connect to the target, and reset it. The running state of the target is unchanged.

Note

The options available depend on the target that you are connecting to.

Making numbered connections

The connection numbers are positive integers that identify elements in the board file. Numbers start at 1 and increment sequentially. However, if you want to use numbered connections with the `CONNECT` command, it is important to understand how these numbers are allocated.

A distinction is made between *access-provider connections* and *endpoint connections*. For example, an *access-provider connection* might be an RVI-ME interface unit while an *endpoint connection* might use RVI-ME to connect to an ARM926EJ™-S processor, shown in Figure 2-2 on page 2-99.

All possible access-provider connections in the current board file are enumerated first, followed by the endpoint connections. Some access-provider connections support many endpoints, possibly even a variable number of them. Because of this, endpoint connections are not allocated an `id` until you `CONNECT` to the access-provider for that endpoint connection and so enable the connections provided by the access-provider. When an access-provider connection is enabled, the endpoint connection `id`s are revised, with numbers allocated to each of the known endpoints.

For example, your board file might include the following access-providers, shown in Figure 2-2 on page 2-99:

- `RVI`
- `RVI-ME`

To connect to an RVI-ME:

```
connect,route 1
```

This command enables the access-provider connection and expands it in the Connection Control window. (Entering the same command again disables the connection and collapses it.)

To connect to a target using RVI-ME, use CONNECT again without the `route` qualifier:

```
connect 2
```

Making named connections

You can use named connections using similar principles to the numbered technique described in *Making numbered connections* on page 2-100. However, you can use named connections to connect to a target whose access-provider connection is not currently enabled or to specify targets where there might be ambiguity (see *Making named endpoint connections* for details).

To enable or disable a route, you enter the route name with an `@` prefix, for example:

```
connect,route @RVI-ME
```

Which is the same as:

```
connect,route 1
```

Note

If you specify a target that has not been configured, you are prompted to configure the target before the access-provider connection is enabled.

Now you can connect to a named target:

```
connect @ARM926EJ-S_0
```

Making named endpoint connections

You can use named connections to connect to a named target where the access-provider is not currently enabled or if you do not know the connection `ids`. Specify the full target name in prefix notation. To connect to the RVI-ME target:

```
connect @ARM926EJ-S_0@RVI-ME
```

This command connects to the ARM926EJ-S on the RVI-ME. If the access-provider, in this case RVI-ME, has not been configured with an `ARM926EJ-S_0`, the connection fails with the message `Types of objects in list do not match`. You must configure the target before you connect to it. The target name must be as defined in the board file, that is as it appears in the Connection Control window.

See also

The following commands provide similar or related functionality:

- *ADDBOARD* on page 2-21
- *DISCONNECT* on page 2-124
- *EDITBOARDFILE* on page 2-146
- *RESTART* on page 2-245
- *RUN* on page 2-250.

2.3.27 CONTEXT

The CONTEXT command displays the current context.

Syntax

```
CONTEXT [/F]
```

where:

/F	Displays all contexts (roots).

Description

The CONTEXT command displays the current context. If you are using the GUI, then the context is displayed in the Output pane. The context includes the current root, module, procedure, and line. The context must be in a module with high-level debug information for the line number to be displayed.

CONTEXT runs asynchronously unless it is run in a macro.

Examples

The following example shows how to use CONTEXT using the dhrystone application:

```
> context
At the PC: (0x00008000): ENTRY\__main
Source view: DHRY_1\main Line 78
```

This demonstrates the case where the PC and the current source view do not correspond. In this case, the editor is displaying the beginning of the function main() at line 78, while the pc is at location 0x8000 in the __main(), the routine that calls main().

The following example sets a breakpoint in main() and runs to that breakpoint:

```
> bi \DHRY_1\#98:0
> go
Stopped at 0x000084D0 due to SW Instruction Breakpoint
Stopped at 0x000084D0: DHRY_1\main Line 98
> con
At the PC: (0x000084D0): DHRY_1\main Line 98
```

Now the PC and the source view are synchronized, the form of the message changes.

Finally, the /F form, of CONTEXT displays the Root: specification shown in the following example (see Chapter 1 *Working with the CLI* for more information on root context specifications):

```
> CONTEXT/F
At the PC: (0x000084D0): DHRY_1_1\DHRY_1_C\main Line 98
Root: @dhrystone\\ [SCOPE]
```

See also

The following commands provide similar or related functionality:

- *DOWN* on page 2-131
- *PRINTSYMBOLS* on page 2-224
- *SCOPE* on page 2-252
- *SETREG* on page 2-259
- *UP* on page 2-335.

2.3.28 COPY

The `COPY` command copies a region of memory.

Syntax

```
COPY addressrange, targetaddr
```

where:

addressrange	Specifies the address range to be copied.
targetaddr	Specifies the starting address where the copied memory is placed.

Description

The `COPY` command copies the contents of a specified block of memory to a block of the same size starting at a specified location.

The command copies data from low address to high addresses, without taking account of overlapping source and destination memory regions. You must not rely on this behavior in future versions of the debugger.

`COPY` runs synchronously unless background access to target memory is supported. Use the `WAIT` command to force it to run synchronously.

Examples

The following examples show how to use `COPY`:

```
copy 0x8100..0x81FF,0x8700
```

Copies the contents of memory at `0x8100` to `0x81FF` to memory at `0x8700` to `087FF`.

```
copy 0x8100..+128,0x8700
```

Copies the contents of memory at `0x8100` to `0x817F` to memory at `0x8700` to `0877F`.

See also

The following commands provide similar or related functionality:

- *COMPARE* on page 2-96
- *FILL* on page 2-161
- *LOAD* on page 2-189
- *READFILE* on page 2-235

- *TEST* on page 2-285
- *SETMEM* on page 2-257.

2.3.29 DBOARD

DBOARD is an alias of DTBOARD (see page 2-133).

2.3.30 DBREAK

DBREAK is an alias of DTBREAK (see page 2-135).

2.3.31 DCOMMANDS

The DCOMMANDS command lists the commands available based on vehicle, target processor, and type of connection.

Syntax

```
DCOMMANDS [{,full | ,alias}] [,cmd_class...] [{;windowid | ;fileid}]
```

```
DCOMMANDS [{,full | ,alias}] =specific_cmd [{;windowid | ;fileid}]
```

where:

cmd_class Specifies a class of commands to have details displayed, and can be any of the following:

`status` **or** `display`

to list *status* and *display* commands

`setstatus` **or** `ss`

to list *setstatus* commands

`breakcomplex` **or** `bc`

to list *breakcomplex* commands

If no command class is specified, all of the commands known to DCOMMANDS are described.

`alias` Show a summary of names and aliases for the specified command class.

`full` Show more detailed information on the specified command class.

specific_cmd Specifies a particular command to display, or `all` to display all commands known to DCOMMANDS.

`;`*windowid* `|` `;`*fileid*

Identifies the window or file where the command is to display its output. This must be a user-defined ;*windowid* or ;*fileid*. See *Window and file numbers* on page 1-5 for details.

If you do not supply a ;*windowid* or ;*fileid* parameter, output is displayed on the screen. If you are using the GUI, then the output is displayed in the Output pane.

Description

The DCOMMANDS command displays the list of commands supported by the current target. The optional command class qualifier enables you to display one or more specific classes of commands. The *specific_cmd* argument shows a specified command. The full qualifier provides extended detail on the command.

Note

Some commands are not listed in the DCOMMANDS command list, and DCOMMANDS reports that these commands are unknown if you request help with the *specific_cmd* argument. This is a limitation of the current implementation of the help system and does not indicate a fault in the operation of the commands.

Examples

The following examples show the use of DCOMMANDS. The first command displays a summary of all status commands that are available on the current target:

```
> dcom,status =all
    dcommands [{,cmd_classes...}] [=specific_cmd] [;viewport]
or  dhelp    [{,cmd_classes...}] [=specific_cmd] [;viewport]
    dtboard  [={resource,...}] [;viewport]
or  dboard   [={resource,...}] [;viewport]
    dtprocess [={task,...}] [;viewport]
or  dvprocess [={task,...}] [;viewport]
    dtfile   [={value,...}] [;viewport]
or  dvfile   [={value,...}] [;viewport]
or  dmap     [={value,...}] [;viewport]
    dtbreak  [={threads,...}] [;viewport]
or  dbreak   [={threads,...}] [;viewport]
```

Note

;viewport in the command syntax can be either ;*windowid* or ;*fileid*.

This command displays a more complete summary of the XTRIGGER command:

```
> dcom,full xtrig
    xtrigger [{,qualifier...}] [={boards,...}]

Qualifiers:
  in_disable  in_enabl  eout_disable  out_enable  onhost

This command is used to set the cross-triggering state of the selected
```

```
boards. This can be used to control what happens when any board stops. It
will be implemented using hardware when possible but can be forced to use
software (on host) methods.
```

Alias

`DHELP` is an alias of `DCOMMANDS`.

See also

The following commands provide similar or related functionality:

- *HELP* on page 2-178
- *SHOW* on page 2-269.

2.3.32 DEFINE

The `DEFINE` command creates a macro for use by other RealView Debugger components.

Note

Because a macro definition requires multiple lines, you cannot use the `DEFINE` command from the RealView Debugger command prompt. Instead, you must either:

- Use the macro command GUI. See the macros chapter in the *RealView Developer Kit v2.2 Debugger User Guide* for more information.
- Write your macro definition in a text file and load it into RealView Debugger using the `INCLUDE` command (see *Macro language* on page 1-7).

Syntax

```
DEFINE [/R] [return_type] macro_name ([parameters])[parameter_definitions] {
macro_body} .
```

where:

`/R` The new macro can replace an existing symbol with the same name.

`return_type` Specifies the return type of the macro. If a type is not specified, *`return_type`* defaults to type `int`.

`macro_name` Specifies the name of the macro.

`parameters` Lists parameters (comma-separated list within parentheses). These parameters can be used throughout the macro definition and are later replaced with the values of the actual parameters in the macro call.

`param_definitions`

Defines the types of the variables in *`parameter_list`*. If types are not specified, the default type `int` is assumed.

`macro_body` Represents the contents of the macro, and is split over many lines. The syntax for *`macro_body`* is:

```
[local_definitions]
macro_statement;[macro_statement;] ...
```

`local_definitions` are the variables used within the `macro_body`.

A *`macro_statement`* is any legal C statement except switch and goto statements, or a debugger command. If *`macro_statement`* is a debugger command, it must start with a dollar sign (`$`) and end with a dollar sign and a semicolon (`$;`). All statements are terminated by a semicolon.

The macro_body ends with a line containing only a period (full stop).

Description

The definition contains a macro name, the parameters passed to the macro, the source lines of the macro, and a terminating period as the first and only character on the last line.

After a macro has been loaded into RealView Debugger, the definition is stored in the symbol table. If the symbol table is recreated, for example when an image is loaded with symbols, any macros are automatically deleted. The number of macros that can be defined is limited only by the available memory on your workstation.

Macros can be invoked by name on the command line where the name does not conflict with other commands or aliases and the return value is not required. You can also invoke a macro on the command line using the MACRO command, and in expressions, for example using the CEXPRESSION command.

Macros can also be invoked as actions associated with:

- a window, for example VMACRO
- a breakpoint, for example BREAKEXECUTION
- deferred commands, for example BGLOBAL.

Note

Macros invoked as associated actions cannot execute GO, or GOSTEP, or any of the stepping commands, for example STEPINSTR.

If you require a breakpoint that, when the condition is met, does something and then continues program execution, you must use the breakpoint continue qualifier, or return 1 from the macro call, instead of the GO command. See the breakpoint command descriptions for more details.

Examples

The following examples show how to use DEFINE:

```
define float square(f)
 float f;
{
  return (f*f);
}
.

define show_i()
{
  int i;
  i = 10;
  $printf "value of i = %d\n", i$;
  return (1);
}
.

define /R int userPrompt()
{
  char userPromptBuffer[100];
  int retval;
  retval = prompt_text( "Please enter text", userPromptBuffer );
  if (retval == 0) {
    $printf "Clicked OK\n"$;
    $printf "%s\n", userPromptBuffer$;
  } else
    $printf "Clicked Cancel\n"$;
  return 1;
}
.
```

See also

The following commands provide similar or related functionality:

- *ALIAS* on page 2-25
- *BGLOBAL* on page 2-38
- *BREAKEXECUTION* on page 2-52
- *CEXPRESSION* on page 2-91
- *GOSTEP* on page 2-175
- *INCLUDE* on page 2-181
- *MACRO* on page 2-195.
- *SHOW* on page 2-269
- *VMACRO* on page 2-341.

2.3.33 DELBOARD

The DELBOARD command deletes a board entry from the displayed list.

Syntax

```
DELBOARD [=resource,...]
```

where:

resource	Identifies the board that is to have its entry deleted from the list.

Description

Use this command to delete a specified non-connected board entry. If you do not specify a board, all non-connected board entries are deleted. You supply the number or name of the board, or 0 for all. This does not affect the file stored on disk, only what is shown.

Example

The following example shows how to use DELBOARD:

```
delboard =6
```

This command deletes the connection definition numbered 6, that must be unconnected when the command is issued. See *CONNECT* on page 2-98 for more information about connection numbers. The deleted connection definition becomes available again when a READBOARDFILE command is issued or the debugger is restarted.

See also

The following commands provide similar or related functionality:

- *ADDBOARD* on page 2-21
- *CONNECT* on page 2-98
- *EDITBOARDFILE* on page 2-146.

2.3.34 DELETE

The `DELETE` command deletes macros or one or more symbols from the symbol table.

Syntax

```
DELETE {symbol_name | \\ | \ | macroname} [,y]
```

where:

symbol_name	Specifies the symbol to be removed from the symbol table.
symbol_name\	Deletes the specified symbol and all symbols it owns (its child symbols).
root\\	Deletes all symbols of the specified root.
\\	Deletes all user-defined symbols of the base root.
\	Deletes all symbols of the current root.
macroname	Deletes the specified macro.
y	Specifies that `DELETE` can delete child symbols if the specified symbol has them. If this is not done, `DELETE` prompts for confirmation before deleting child symbols.

Description

The `DELETE` command deletes symbols from the symbol table associated with the current connection. Symbols are entered into the symbol table when an executable file containing them is loaded onto the connection using `LOAD` or `RELOAD`, and when you use the `ADD` command.

Deleting a symbol or group of symbols is useful if the program has changed, perhaps as a result of runtime patching of the executable. To change the memory location of a symbol such as an address label, you must first delete it and then add it again at the new location.

You can also use the `DELETE` command to delete debugger macros that you have created using the `MACRO` command.

You cannot use `DELETE` to delete debugger command aliases. Instead, define the alias to be nothing:

```
alias name=
```

Rules

The following rules apply to the use of the `DELETE` command:

- The `DELETE` command runs asynchronously unless in a macro.
- All debugging information for that symbol is deleted, but program execution is unchanged.
- Only program symbols, macros, and user-defined debugger symbols can be deleted from the symbol table. Predefined symbols, such as register names, cannot be deleted.
- If the specified symbol or macro has local symbols, confirmation is requested that you want to delete all the local symbols. Entering the `,y` parameter provides this confirmation automatically.

See also

The following commands provide similar or related functionality:

- *ADD* on page 2-18
- *ALIAS* on page 2-25
- *PRINTSYMBOLS* on page 2-224
- *DEFINE* on page 2-110.

2.3.35 DELFILE

The DELFILE command removes filenames from the executable file list, provided the specified file is not loaded onto the target.

Syntax

```
DELFILE [,auto] {filename | file_num}
```

where:

auto	Causes the command to remove unloaded files from the file list that were added as a result of the ADDFILE,auto command.
filename|*file_num*	Identifies a file to be removed from the executable file list. You can include one of more environment variables in the filename. For example, if MYPATH defines the location C:\Myfiles, you can specify: `delfile "$MYPATH\\myimage.axf"`

Description

The ADDFILE and the DELFILE commands are used to manipulate the executable image file list. This list is in most cases only one file, the executable you load onto the target using LOAD. There are circumstances where you must load more than one file onto the target at once. In these cases you use ADDFILE to set up the files to load, and RELOAD or LOAD/A to load them onto the target.

You use DELFILE to remove unloaded files that you have added to the executable file list. There are several ways to specify the files to delete:

- by complete filename, for example C:\Source\dhry\Debug\dhry.axf
- by short filename, for example dhry.axf
- by file number, for example 2
- as the currently unloaded files that were added to the list by ADDFILE,auto
- as all currently unloaded files.

DELFILE with no arguments deletes all currently unloaded files, and DELFILE,auto deletes any currently unloaded files added as a result of an ADDFILE,auto.

Use DTFILE to display the current file list, including the defined short filenames, file numbers and whether the file is loaded or not.

Note

- If you use the full filename you must enclose it in double quotes. You do not have to quote the short filename in quotes, although you can.
- You cannot delete multiple named or numbered files in a single command. Use multiple `DELFILE` commands, or delete all files and then use `ADDFILE` as required.
- An executable file must be unloaded from the target before its name can be removed from the file list. Use the `UNLOAD` command to unload a file that is no longer being used by the target.

Examples

The following examples show how to use `ADDFILE` and `DELFILE`:

```
> addfile ="C:\Source\helloworld\Debug\helloworld.axf"
> dtfile
File 1 with modid <not loaded>: Symbols not Loaded. 0 Sections.
'helloworld.axf' As 'C:\Source\helloworld\Debug\helloworld.axf'
1,1/* ARM7TDMI  PC=0x00008000   !"C:\Source\helloworld\Debug\helloworld.axf"
```

A file is added to the executable list, using `ADDFILE`, and `DTFILE` shows that it is on the list and has file number, or `id`, of 1 (the `File 1` part of the output from `DTFILE`).

Because the file has not been loaded, the debugger has not read the symbol table to determine the code, data and *Base Stack Segment* (BSS) section sizes that a `DTFILE` following a `LOAD` displays. See `DTFILE` on page 2-137 for more information.

To delete this file, you can use the file ID, reported in the first line of `DTFILE` output, as follows:

```
> delfile 1
> dtfile
No files for this process.
```

The `DTFILE` output tells you that the deletion was successful. In this particular case, the file `id` is not required, because a `DELFILE` with no parameters deletes all unloadable files. For example:

```
> addfile ="C:\Source\helloworld\Debug\helloworld.axf"
> delfile
> dtfile
No files for this process.
```

You can name the file to delete, using either the full name of the file or the short name listed in the `DTFILE` result:

```
> addfile ="C:\Source\helloworld\Debug\helloworld.axf"
> delfile helloworld.axf
> dtfile
No files for this process.

> addfile ="C:\Source\helloworld\Debug\helloworld.axf"
> delfile "C:\Source\helloworld\Debug\helloworld.axf"
> dtfile
No files for this process.
```

See also

The following commands provide similar or related functionality:

- *ADDFILE* on page 2-23
- *DTFILE* on page 2-137
- *LOAD* on page 2-189
- *RELOAD* on page 2-239
- *UNLOAD* on page 2-333.

2.3.36 DHELP

`DHELP` is an alias of `DCOMMANDS` (see page 2-107).

2.3.37 DISABLEBREAK

The `DISABLEBREAK` command disables one or more specified breakpoints.

Syntax

```
DISABLEBREAK [,h] [break_num,...]
```

where:

break_num	Specifies one or more breakpoints to disable, separated by commas. You identify breakpoints by their position in the list displayed by the `DTBREAK` command (see page 2-135).
`h`	Do not use this qualifier. It is for debugger internal use only.

Description

The `DISABLEBREAK` command disables one or more breakpoints. A disabled breakpoint is removed from the target as if the breakpoint were deleted, but the debugger keeps a record of it. You can then enable it again by referring to the breakpoint number when required, rather than having to recreate it from scratch.

If you issue the command with no parameters then all breakpoints for this connection are disabled. Disabling a breakpoint that is already disabled has no effect.

Examples

The following examples show how to use `DISABLEBREAK`:

`disablebreak 4,6,8`

Disables the fourth, sixth, and eighth breakpoints in the current list of breakpoints.

`disablebreak` Disables all the current breakpoints.

See also

The following commands provide similar or related functionality:

- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78

- *CLEARBREAK* on page 2-94
- *DTBREAK* on page 2-135
- *CLEARBREAK* on page 2-94
- *ENABLEBREAK* on page 2-150.

2.3.38 DISASSEMBLE

The `DISASSEMBLE` command displays memory addresses and corresponding assembly code on the **Dsm** tab page of the Code window.

Syntax

`DISASSEMBLE [{/D|/S|/A|/B}] [{address | @stack_level}]`

where:

`/D`	Disassemble using the default instruction format. For ARM architecture processors, this is ARM state.
`/S`	Disassemble using the standard instruction format. For ARM architecture processors, this is ARM state.
`/A`	Disassemble using the alternate instruction format. For ARM architecture processors, this is Thumb state.
`/B`	Disassemble using bytecodes on ARM architecture processors.
address	Specifies the starting address for disassembly. This can be a literal address or a debugger expression.
stack_level	Enables you to specify the starting point without knowing its address. Stack level 0 is the current address in the current procedure, stack level 1 is the code address from which the current procedure was called.

Description

The `DISASSEMBLE` command displays memory addresses in hexadecimal and assembly code on the **Dsm** tab page of the Code window, starting at the specified memory location and using the assembler mnemonics and register names associated with the processor type of this connection.

Where multiple assembler mnemonics exist for the same processor type (for example, with the ARM and the GNU assemblers for ARM processors) the debugger can only use one of them. There is no way to select the alternate form.

——— **Note** ———

Different target connections can be connected to different processor types and so have differing register names and assembler mnemonics.

————————

If the specified address falls in the middle of an instruction, the whole instruction is displayed. Memory is displayed starting at the address held in the PC if you do not supply an address. The current execution context and variable scope of the program remains unchanged even if you select an alternate stack level.

If you issue the `OPTION` command with the `LINES=ON` option, source code is intermixed with the assembly language code. If you issue the `OPTION` command with the `SYMBOLS=ON` option, symbol references are displayed with the assembly language symbols and labels.

The `DISASSEMBLE` command runs synchronously unless background access to target memory is supported. Use the `WAIT` command to force it to run synchronously.

Examples

The following examples show how to use `DISASSEMBLE`:

`DISASSEMBLE /S @1`

Disassemble, using the standard instruction format (for ARM processors, the ARM state format), the instructions that are executed when the current function returns, displaying the result in the **Dsm** tab in the File Editor pane.

`DISASSEMBLE 0x80200`

Disassemble, using an instruction format selected using symbol table information, the instructions starting at address `0x80200`, displaying the result in the **Dsm** tab in the File Editor pane.

See also

The following commands provide similar or related functionality:

- *DUMPMAP* on page 2-143
- *DUMP* on page 2-141
- *LOAD* on page 2-189
- *MEMWINDOW* on page 2-201
- *MODE* on page 2-203
- *PRINTDSM* on page 2-218
- *WHERE* on page 2-348.

2.3.39 DISCONNECT

The `DISCONNECT` command disconnects the debugger from a target.

Syntax

```
DISCONNECT [,all | ,gui] [=][{targetid | @targetname}]
```

where:

`all`	Disconnects all connections.
`gui`	Enables you to choose the disconnect mode from a dialog or prompt: • If you use this option in the GUI CLI, it displays a dialog. • If you use this option in the headless debugger, it displays a prompt. The disconnect specifies what state you want the debugger to leave the target in after the disconnection. See *Disconnect modes* on page 2-125 for more details.
`targetid`	Specifies the required target as a number.
`targetname`	Specifies the required target as a name as it appears in the GUI.

Description

The `DISCONNECT` command disconnects the debugger from a target, undoing the action of a previous `CONNECT`. You can specify the target using the numeric or textual methods outlined for the `CONNECT` command.

Note

If you set the disconnect mode in the board (`.BRD`) file of the target, or the `.BCD` file for a board, the target disconnects using that mode. If you specify `prompt` for the disconnect mode, then the `DISCONNECT` command acts as though you specified the `,gui` qualifier. See the *RealView Developer Kit v2.2 Debugger User Guide* for more details.

The `DISCONNECT` command runs asynchronously.

For more information see *CONNECT* on page 2-98 for details about connection `ids` and named connections.

Disconnect modes

When you disconnect from a target, the disconnect mode determines what happens to the target:

As-is with Debug Leave the target in the current run state and the current debug state. That is:

- If the target is running now, leave it running. If the target is stopped in debug state now, leave it stopped.
- Current debug state intact, for example, breakpoints remaining.

As-is without Debug

Leave the target in the current run state but without the current debug state. That is:

- If the target is running now, leave it running. If the target is stopped in debug state now, leave it stopped.
- Current debug state lost, for example, breakpoints removed.

Note

The options available depend on the target that you are disconnecting from.

Implications for RTOS connections

If you disconnect from an RTOS connection, RealView Debugger sends a command to the Debug Agent, which might resume all stopped threads depending on how the Debug Agent is implemented.

Examples

The following examples show how to use `DISCONNECT`:

`disconnect,all`	Disconnect all currently connected connections.
`disconnect`	Disconnect the current target (the target shown in the title bar).
`disconnect =7`	Disconnect the target with a connection `id` of `7`.
`disconnect,gui 7`	Display the Disconnect Mode selection box to disconnect the target with a connection `id` of 7.

See also

The following commands provide similar or related functionality:

- *ADDBOARD* on page 2-21
- *CONNECT* on page 2-98
- *RESTART* on page 2-245.

2.3.40 DLOADERR

The `DLOADERR` command displays possible reasons for the last load error.

Syntax

```
dloaderr [{,gui | ;windowid | ;fileid}]
```

where:

`gui` This qualifier causes the results to be displayed in a dialog.

Note

This qualifier has no effect in the headless debugger.

`;windowid | ;fileid`

This parameter identifies the window in which you want the command to display its output. This must be a user-defined `;windowid` or `;fileid`. See *Window and file numbers* on page 1-5 for details.

Description

The `DLOADERR` command displays possible reasons for the most recent program executable load error, and suggests actions you might take.

If you issue the command with no qualifier or parameter, then its output is displayed on the screen. If you are using the GUI, then the output is displayed in the Output pane. For more information on redirecting message output see *VOPEN* on page 2-343.

See also

The following commands provide similar or related functionality:

- *LOAD* on page 2-189
- *RELOAD* on page 2-239.

2.3.41 DMAP

`DMAP` is an alias of `DTFILE` (see page 2-137).

2.3.42 DOS_resource-list

Displays an RTOS resource list or shows details of one element in that list.

Syntax

DOS_*resource-list* ,*qualifier* [=*value*] [{;*windowid* | ;*fileid*}]

where:

resource-list

Specifies the resource list.

qualifier	Specifies what to display, that is `all` or `detail`. `detail` is the default if you specify a *value*. If you do not specify a *value*, you must use `all`.
value	Identifies an object in the specified resource list.

;*windowid* | ;*fileid*

Specifies a window or file where the details are to be sent. This must be a user-defined ;*windowid* or ;*fileid*. See *Window and file numbers* on page 1-5 for details.

Description

The DOS_*resource-list* command displays an RTOS resource list or shows details of one element in that list. If you are using the GUI, then these are displayed in the Output pane. It displays the information as shown in the Details area of the Resource Viewer window. The *resource-list* and *qualifier* depend on the RTOS you are using.

You can get a list of these commands using the `DCOMMANDS` command after stopping or halting the RTOS image. To do this, and save the commands to the file `commands.txt`, enter:

```
fopen 100,'c:\path\commands.txt'
dcommands all ;100
vclose 100
```

You can also determine these from the Resource Viewer window:

- *resource-list* is determined by the tab you select in the Resource List, with the exception of the **Conn** tab
- *qualifier* is determined by right clicking on an object in the selected tab of the Resource List.

You might want to log your use of the Resource Viewer window to determine the CLI commands you can use with your RTOS. See *LOG* on page 2-193 for details. You can then modify the log file, and use it as a command script, see *INCLUDE* on page 2-181.

See the *RealView Developer Kit v2.2 Extensions User Guide* for details on using the Resource Viewer window.

Examples

The following examples show how to use `DOS_`*`resource-list`*:

```
fopen 100,'c:\myfiles\threads.txt' dos_thread_list,all ;100 vclose 100
```

Copies the details of all thread resources to the file `c:\myfiles\threads.txt`.

```
dos_thread_list,detail = thread_4 dos_thread_list,detail = 0x39d8 dos_thread_list
= 0x39d8
```

Displays details about the specified thread.

```
dos_timer_list,detail = 0x39d8
```

Displays details about the specified timer.

See also

The following commands provide similar or related functionality:

- *AOS_resource-list* on page 2-32
- *BREAKINSTRUCTION* on page 2-61
- *GO* on page 2-173
- *HALT* on page 2-177
- *OSCTRL* on page 2-214.
- *STOP* on page 2-283.

2.3.43 DOWN

The `DOWN` command moves the variable scope and source location down the stack (that is, away from the program entry point, towards the current PC).

Syntax

`DOWN [levels]`

where:

levels	Specifies the number of stack levels to move down. This must be a positive number.

Description

This command moves the current variable scope, and source or disassembly view location down the stack by the specified number of levels. The debugger modifies the local variable scope to display the variables in the new location, and potentially hiding those at the previous level.

If you are already at the lowest level (nearest to the program entry point), a message reminds you that you cannot move down any more. You must have used an `UP` command or a `SCOPE` command before a `DOWN` command becomes meaningful. You can move down one level by using the command without parameters.

The `DOWN` command runs synchronously unless background access to target memory is supported. Use the `WAIT` command to force it to run synchronously.

Example

The following example shows how to use `DOWN`. The `UP` command moves the context up the stack to the enclosing function, so that a variable `index` is in scope. The value of the variable is displayed, and it is decided to discover that another variable, `count`, by looking at the preceding function. When count is displayed, the `DOWN 2` command is used to return down the stack two levels, to the scope of the initial function

```
> up
> ce index
index = 3
> up
> ce count
count = 55
> down 2
```

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *CONTEXT* on page 2-102
- *EXPAND* on page 2-157
- *SCOPE* on page 2-252
- *UP* on page 2-335
- *WHERE* on page 2-348.

2.3.44 DTBOARD

The DTBOARD command displays information about the current or a specified board.

Syntax

DTBOARD [="*resource*",...] [{;*windowid* | ;*fileid* }]

where:

resource Identifies the board that is to have its details displayed. If you use a board name, rather than the ID, you must specify each name in double quotes, for example:

```
dtboard ="ARM926EJ-S_0"
```

windowid | *fileid*

Identifies the window in which you want the command to display its output. This must be a user-defined *windowid* or *fileid*. See *Window and file numbers* on page 1-5 for details.

Description

The DTBOARD command displays information about the current or a specified board. If you do not specify a board, the command displays information about the current board. If you do not supply a ;*windowid* parameter, the output is displayed on the screen. If you are using the GUI, then the output is displayed in the Output pane. For more information see *VOPEN* on page 2-343.

Example

The following example shows how to use DTBOARD.

```
> dtboard
Connected Board 'ARM926EJ-S_0' Port 0: Server supporting Single Tasking.
Port string: localhost
Entry of router/broker RVI-ME
```

Alias

DBOARD is an alias of DTBOARD.

See also

The following commands provide similar or related functionality:

- *ADDBOARD* on page 2-21

- *DTFILE* on page 2-137.

2.3.45 DTBREAK

The DTBREAK command displays information on all breakpoints and tracepoints set.

Syntax

DTBREAK [=*thread*,...] [{;*windowid* | ;*fileid*}]

where:

thread	Not supported in this release.

windowid | *fileid*

Identifies the window in which you want the command to display its output. This must be a user-defined *windowid* or *fileid*. See *Window and file numbers* on page 1-5 for details.

If you do not supply a *windowid* or *fileid* parameter, the output is displayed on the screen. If you are using the GUI, then the output is displayed in the Output pane.

Description

The DTBREAK command displays information about the currently defined breakpoints and tracepoints.

Example

The following example shows how to use DTBREAK.

```
> dtbreak
S Type  Address         Count   Miscellaneous
- ----  -------         -----   -------------
  Instr 0x24000408      0
  Read  0x24000434      0
  Trace InstrExec       0x000085A8      0
```

Alias

DBREAK is an alias of DTBREAK.

See also

The following commands provide similar or related functionality:

- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52

- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *CLEARBREAK* on page 2-94
- *DISABLEBREAK* on page 2-120
- *DTRACE* on page 2-139
- *ENABLEBREAK* on page 2-150
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.46 DTFILE

The `DTFILE` command displays information about one or more specified files or all files of the current process.

Syntax

```
DTFILE [=file_num,...][{;windowid | ;fileid}]
```

where:

`file_num` One or more integer numbers that identify the file or files about which you want to see information. If you do not supply this parameter, details of all the currently loaded files are displayed.

`windowid | fileid`

Identifies the window or file where the command is to direct its output. This must be a user-defined *`windowid`* or *`fileid`*. See *Window and file numbers* on page 1-5 for details.

If you do not supply a *`windowid`* or *`fileid`* parameter, output is displayed on the screen. If you are using the GUI, then the output is displayed in the Output pane.

Description

The `DTFILE` command displays information about the currently loaded executable file. The file numbers are the same as those used in the `ADDFILE` and `DELFILE` commands. The information displayed varies:

- if the file has been loaded onto the target, then the information contains details about the code and data section sizes and the load addresses
- if the file has not been loaded, the debugger has not yet determined the code and data sizes and so does not display them.

The first line of the output includes the following information:

File *file_num*

Used by the `ADDFILE`, `DELFILE`, `RELOAD` and `UNLOAD` commands to refer to the file.

modid *num* An internal number.

Symbols Loaded

This item tells you whether the executable file has program debug symbols and whether they have been loaded. In most cases you require debug symbols to make sense of the program instructions.

***n* sections** This item tells you how many program sections there are in the file. Each loaded program section is normally listed with any associated information.

The second line of output contains first the shortname and then the file path name of the file. The short name is an abbreviation of the name, normally the filename with no directory specification. The file path name includes the full directory path name for the file. You must normally specify the file path name enclosed in double quotes when entering it in commands.

If a file was built with separate load regions defined, these load regions are also shown in the output.

Example

The following example illustrates the output of `DTFILE`, displaying information about a loaded executable called `shapes.axf`.

```
> dtfile =1
File 1 with modid 1: Symbols Loaded. 2 Sections.
  'shapes.axf' As 'c:\src\cpp\shapes_Data\DebugRel\shapes.axf'
  Code section of size 10732 at 0x00008000: ER_RO
  BSS  section of size   356 at 0x0000A9EC: ER_ZI
```

Alias

`DMAP` and `DVFILE` are aliases of `DTFILE`.

See also

The following commands provide similar or related functionality:

- *ADDFILE* on page 2-23
- *LOAD* on page 2-189
- *MEMMAP* on page 2-197
- *RELOAD* on page 2-239
- *UNLOAD* on page 2-333.

2.3.47 DTRACE

The `DTRACE` command shows information on trace.

Syntax

<u>`dtrace`</u> `[{;`*`windowid`*` | ;`*`fileid`*`}]`

where:

`windowid | fileid`

Identifies the window in which you want the command to display its output. This must be a user-defined *`windowid`* or *`fileid`*. See *Window and file numbers* on page 1-5 for details.

If you do not supply a *`windowid`* or *`fileid`* parameter, output is displayed on the screen. If you are using the GUI, then the output is displayed in the Output pane.

Description

The `DTRACE` command displays information about the trace analyzer you are using and the triggers that are defined.

Example

The following example illustrates the output of `DTRACE`:

```
> dtrace
ARM Analyzer: ARM Embedded Trace Macrocell. Version 1.0.
2 Tracepoints defined.
  Trigger On at Code 0x846C.
  Trigger On at Code 0x8540.
Buffer collected Before Trigger.
(Before/Around/AfterSupported).
```

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307

- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.48 DUMP

The `DUMP` command displays memory contents in hexadecimal or ASCII format.

Syntax

```
DUMP [{/B|/H|/W|/8|/16|/32}] [{address | address_range}]
```

where:

`/B, , /8`	Sets the display format to byte (8 bits). If the processor naturally addresses bytes (for example, ARM7TDMI) then this is the default setting.
`/H, , /16`	Sets the display format to halfword (16 bits).
`/W, , /32`	Sets the display format to word (32 bits).
`address`	Specifies a memory address at which to begin the display of contents. The remainder of that 16-byte line and the whole of the following 16-byte line are displayed.
`address_range`	Specifies a range of memory addresses whose contents are to be displayed. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

The `DUMP` command displays memory contents in bytes, words or long words as hexadecimal and ASCII characters on the screen. If you are using the GUI, then they are displayed in the Output pane.

If you do not specify any parameters, the next five lines of data after the previously dumped address range are displayed. In the character output format, nonprintable characters (such as a carriage return) are represented by a period (`.`).

The `DUMP` command runs synchronously unless background access to target memory is supported. Use the `WAIT` command to force it to run synchronously.

Example

The following example illustrates the output of `DUMP`. The first example displays two rows of memory from `0x8000`.

```
> dump 0x8000
  0x00008000 00 00 00 EA 24 06 00 EA  28 C0 8F E2 00 0C 9C E8 ....$...(.......
  0x00008010 0C A0 8A E0 01 70 4A E2  0C B0 8B E0 0B 00 5A E1 .....pJ.......Z.
```

Executing DUMP again displays a page of memory from 0x8020.

```
> dump
             00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
             ------------------------------------------------
  0x00008020 1D 06 00 0A 0F 00 BA E8  14 E0 4F E2 01 00 13 E3 ..........O.....
  0x00008030 03 F0 47 10 03 F0 A0 E1  6C 6B 00 00 7C 6B 00 00 ..G.....lk..|k..
  0x00008040 00 40 A0 E3 00 50 A0 E3  00 60 A0 E3 00 C0 A0 E3 .@...P...`......
  0x00008050 10 20 52 E2 70 10 A1 28  FC FF FF 8A 82 2E B0 E1 . R.p..(........
  0x00008060 30 00 A1 28 00 C0 81 45  0E F0 A0 E1 04 30 9F E5 0..(...E.....0..
```

Requesting a DUMP of words of memory, and specifying a range of addresses produces the following result:

```
dump /h 0x9004..0x9012
  0x00009000             0001  E1A0    1004  E28D  0780  EB00     ...........
  0x00009010 0000  E350                                       ..P.
```

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *FILL* on page 2-161
- *MEMWINDOW* on page 2-201
- *WRITEFILE* on page 2-351.

2.3.49 DUMPMAP

The `DUMPMAP` command writes the current memory map out as a file, using the native linker format.

Syntax

```
DUMPMAP [,gui] filename
```

where:

`gui` If an error occurs when executing the command or when the breakpoint is triggered, the GUI is used to report it.

Otherwise, the error is reported to the command pane.

——— **Note** ———

This qualifier has no effect in the headless debugger.

———————————

`filename` Specifies the filename or file pathname to which the map is written. It must be enclosed in double quotes if a pathname is specified, and the pathname must already exist on your system.

You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myfiles`, you can specify:

```
dumpmap '$MYPATH\ld.map'
```

Description

The `DUMPMAP` command writes a linker map file in the format associated with the current processor to the named file.

If the *filename* is a file path name, it must be enclosed in double quotes. If it is not an absolute path name, it is written relative to the current directory of RealView Debugger, which on Windows is normally your desktop.

If the file already exists, RealView Debugger only replaces the information between the `RVDEBUG: generated data block` and the `RVDEBUG: generated data above` comments.

The command runs synchronously.

Example

The following command shows the output of `DUMPMAP` when run on an ARM architecture processor, writing information about the `dhrystone` example program to the file `c:\source\ld.map`:

```
dumpmap "c:\source\ld.map"
```

Example 2-1 shows an example of what the map file contains.

Example 2-1 ARM Architecture linker map file output

```
/* Linker Command file for the ARM processor */
/* This file was generated by RVDEBUG. You can edit everything
   outside the MEMORY block defined by RVDEBUG. Updates by
   RVDEBUG will only affect that block. */

/* RVDEBUG: generated data block.
   Do not modify this block. Do not put MEMORY lines above
   this line, put below end of this block. */
MEMORY
{
  A_RAM:        org=0x1000000, len=0x6000000  /* external 'Sect dhry.axf,dhry.axf,dhry.axf' */
  A_RAM1:       org=0x7000004, len=0xFFFFFC  /* external 'Sect Stack' */
}
/* RVDEBUG: generated data above */
```

See also

The following command provides similar or related functionality:

- *MEMMAP* on page 2-197.

2.3.50 DVFILE

`DVFILE` is an alias of `DTFILE` (see page 2-137).

2.3.51 EDITBOARDFILE

The `EDITBOARDFILE` command enables you to edit a specified board file.

Note

This command is not available in the headless debugger.

Syntax

```
EDITBOARDFILE [,configure] [="boardfilename","routeID"...]
```

where:

`configure` Opens the configuration dialog for the debug target.

`boardfilename`

Identifies the board file that you want to edit. This must be in double quotes, for example, `"myboard.brd"`.

You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myfiles`, you can specify:

```
editboardfile ="$MYPATH\\myboard.brd"
```

`routeID` Identifies the route ID for the target associated with the board file that you want to edit. This must be in double quotes, for example, `"3"`.

Description

The `EDITBOARDFILE` command displays the Connection Properties window to edit the specified board file. If you do not specify a board file, the settings of the current board file are displayed for you to edit.

Note

If you specify a *`routeID`*, you must also specify a blank *`boardfilename`*, for example:

```
editboardfile,configure "","3"
```

You can specify one or more *`boardfilename`*/*`routeID`* combinations.

If you make any changes to a board file, the updated file is reread when you close the Connection Properties window.

The command runs asynchronously.

Example

The following example shows how to use EDITBOARDFILE to edit the Connection Properties dialog:

```
editboardfile
```

The following example shows how to use EDITBOARDFILE to edit the board file for a specific connection:

```
editboardfile,configure "","4"
```

If you specify a board that does not have a device-specific configuration, then the Connection Properties dialog is displayed.

See also

The following commands provide similar or related functionality:

- *ADDBOARD* on page 2-21
- *DTBOARD* on page 2-133
- *READBOARDFILE* on page 2-234.

2.3.52 EMURESET

The `EMURESET` command tests and resets a hardware emulator.

Syntax

```
EMURESET [,test] id
```

where:

`test`	Runs an emulation test on the connection identified by *`id`*. This can involve JTAG testing or self checks.
`id`	Connection identity.

Description

The `EMURESET` command resets a hardware emulator or monitor. This is not the same as `RESET` which resets the target processor or board. The emulation reset is used to set the communications up properly or to prepare the board for debugging.

Example

The following example shows how to reset the hardware on the connection with an ID of 4.

```
emureset 4
```

Alias

`EMURST` and `HWRESET` are aliases of `EMURESET`.

See also

The following commands provide similar or related functionality:

- *RESET* on page 2-241
- *RESTART* on page 2-245
- *WARMSTART* on page 2-347.

2.3.53 EMURST

`EMURST` is an alias of `EMURESET` (see page 2-148).

2.3.54 ENABLEBREAK

The ENABLEBREAK command enables one or more specified breakpoints.

Syntax

```
ENABLEBREAK [,h] [break_num,...]
```

where:

break_num	Specifies one or more breakpoints to enable, separated by commas. You identify breakpoints by their position in the list displayed by the DTBREAK command (see page 2-135).
h	Do not use this qualifier. It is for debugger internal use only.

Description

The ENABLEBREAK command enables one or more breakpoints that have been disabled. A disabled breakpoint is removed from the target as if the breakpoint were deleted, but the debugger keeps a record of it. You can enable it again, using this command, by referring to the breakpoint number, avoiding then having to recreate it from scratch.

If you issue the command with no parameters then all breakpoints are enabled. Enabling a breakpoint that is already enabled has no effect.

The command runs synchronously.

Example

The following examples show how to use ENABLEBREAK:

`enablebreak 4,6,8`	Enables the fourth, sixth, and eighth breakpoints in the current list of breakpoints.
`enablebreak`	Enables all the current breakpoints.

See also

The following commands provide similar or related functionality:

- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *CLEARBREAK* on page 2-94

- *DISABLEBREAK* on page 2-120
- *DTBREAK* on page 2-135
- *RESETBREAKS* on page 2-243.

2.3.55 ERROR

The `ERROR` command specifies what happens if an error occurs in processing an `INCLUDE` file.

Syntax

```
ERROR = {quit | abort | continue}
```

where:

`quit`	Instructs the debugger to quit the session and exit to the operating system.
`abort`	Instructs the debugger to return to command mode and wait for keyboard input.
`continue`	Instructs the debugger to abandon this command and execute the next command in the include file.

Description

The `ERROR` command specifies the action the debugger takes if an error occurs while processing an include file. If you issue the `ERROR` command without parameters, program execution terminates.

The `ERROR` command runs asynchronously unless in a macro.

Example

The following example shows how to use `ERROR`:

`error = abort`	If an error occurs, abort reading the include file and return to the command prompt.

See also

The following commands provide similar or related functionality:

- *INCLUDE* on page 2-181
- *QUIT* on page 2-233.

2.3.56 ETM_CONFIG

The `ETM_CONFIG` command provides control over the ARM ETM.

Syntax

```
ETM_CONFIG  [,qualifier...]
```

where:

qualifier	Is a list of qualifiers. The possible qualifiers are described in *List of qualifiers*.

Description

The `ETM_CONFIG` command provides control over the ARM ETM. The arguments to a single invocation of the command specify a configuration of the ETM, so the presence or absence of qualifiers is relevant.

For more information on the terms used by the ETM and options it provides, see the *Embedded Trace Macrocell™ Specification* and the chapter that describes tracing in the *RealView Developer Kit v2.2 Extensions User Guide*.

List of qualifiers

The list of qualifiers is dependent on the processor and vehicle. The command handler generates an error if a specific combination is invalid for a specific processor or vehicle, but this is determined when you issue the command. The possible qualifiers are:

`size:`*n*	Set the ETM trace buffer size to *n* records.
`mmap_decode:`*n*	Set the ETM memory map value to *n*.
`FIFO_hw:`*n*	Set the FIFO high-water mark to *n*.
`port_width:`*n*	Set the ETM port width, where *n* is one of:

`0`	4-bit port.
`1`	8-bit port.
`2`	16-bit port.
`3`	24-bit port.
`4`	32-bit port.

The 24-bit and 32-bit settings are supported only for ETB11 connections using RealView ICE.

`time_stamps`	Enable time stamping if the ETM and trace capture hardware support it. To disable, issue the command without this qualifier.
`stall_full`	Enable processor stalling if the FIFO becomes full, if the ETM and processor support it. To disable, issue the command without this qualifier.
`half_rate`	Enable half-rate clocking of the trace port by the ETM. For full-rate, issue the command without this qualifier.
`cycle_accurate`	Enable cycle-accurate tracing, if the ETM supports it. To disable, issue the command without this qualifier.
`coprocessor`	Enable coprocessor tracing. To disable, issue the command without this qualifier.
`disableport`	Disable the ETM trace port. To enable, issue the command without this qualifier.
`suppressdata`	Suppress data tracing if the FIFO becomes full. To leave data tracing enabled, issue the command without this qualifier.
`nomultiplex`	Select the normal (not multiplexed or demultiplexed) trace port transmission mode.
`multiplex`	Select the multiplexed trace port transmission mode.
`demultiplex`	Select the demultiplexed trace port transmission mode.
`dataonly`	Trace only data bus transfers.
`addronly`	Trace only address bus transfers. (Deprecated)
`fulltrace`	Trace both data and address bus transfers. (Deprecated)
`portratio:`*`n`*	Enables ETM v3 port speed to ETM clock speed ratios to be set. This is supported only by ETM v3. Appropriate values for *n* are:

`0`	Use a 1:1 ratio.
`1`	Use a 1:2 ratio.
`2`	Use a 1:3 ratio.
`3`	Use a 1:4 ratio.
`4`	Use a 2:1 ratio.
`5`	Use dynamic ratio modes for on-chip trace.
`6`	Use the implementation-defined mode, if implemented by the ASIC designer.

`datasuppression`	Enables ETM v3 data suppression on FIFO full. This is supported only by ETM v3.
`filtercoprocessor`	Enables filtering of CPRTs when data trace is enabled. This is supported only by ETM v3.
`packauto`	Selects the automatic packing mode for the TPA.
`packnormal`	Selects the normal packing mode for the TPA.
`packdouble`	Selects the double packing mode for the TPA.
`packquad`	Selects the quad packing mode for the TPA.
`twin`	Selects the connection that the current connection is connected (twinned) with. This is supported only by DCP/TCM hardware.
`twinmaster`	Selects the processor that is to be the DCP master. This is supported only by DCP/TCM hardware.

Examples

The following examples show how to use `ETM_CONFIG`:

```
ETM_CONFIG,port_width:0,coprocessor,fulltrace,size:10240
```

Set up the ETM for a 4-bit, full-rate, nonmultiplexed trace port, no stalling or timestamps, 10K trace records, address and data tracing, and in non cycle-accurate mode.

```
ETM_CONFIG,port_width:1,stall_full,multiplex,fulltrace,suppressdata,size:1024
```

Set up the ETM for an 8-bit, full-rate, multiplexed trace port, processor stalling and data suppression on FIFO full, no timestamps, 1024 trace records, address and data tracing, and in non cycle-accurate mode.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAREAD* on page 2-307

- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAWRITE* on page 2-313
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.57 EXPAND

The EXPAND command displays the values of parameters to a procedure and any local variables that have been set up.

Syntax

`EXPAND [@stack_level] [{,windowid | ,fileid}]`

Where:

@stack_level	Specifies a stack level if you want to see only a single level expanded. For example, you can specify @3 to expand stack level 3 only.
,windowid | ,fileid	Indicates that the output is to be directed to the specified window or file. You must use one of the numbers listed in *Window and file numbers* on page 1-5.

Description

The EXPAND command displays the values of parameters to a procedure and any local variables that have been set up. You can expand any procedure in a directly called chain from the main program to the current procedure. Other procedures are not accessible.

If no stack level is specified, all procedures nested on the stack are displayed. Stack levels are numbered starting with the current procedure equaling 0, the caller of this procedure is 1, the caller of that procedure is 2.

The EXPAND command runs synchronously.

Messages that can be output by the EXPAND command have the following meanings:

`<Bad float>`	Invalid floating-point value, cannot be converted.
`<bad size>`	Type size invalid.
`<UNKNOWN: xx>`	Invalid enum value, where xx = value.
`<INFINITY>`	Floating-point value is infinity.
`<Invalid value (x)>`	Error number (x) occurred.
`<NAN>`	Not a number (for a floating-point value).

`<not a source procedure. Address is ...>`	Routine is not defined as a function in the object file.
`<not alive>`	Local register variable no longer exists.
`<Not in procedure>`	PC located before first executable line.
`<unknown type>`	Type is not recognized by the debugger.

Example

The following example illustrates the `EXPAND` command executed during a run of the `dhrystone` program. You can see three of the messages in use: an `UNKNOWN` enum value, a variable that is not alive, and a procedure that has no source or debug information available.

```
> go
> expand
  00. Proc_1: at line 309.
    Ptr_Val_Par07FFFF60 = (record *)0x01000260
    Next_Record00000005 = (record *)0x0100C274
  01. main: at line 170.
    Int_1_Loc 07FFFF60 = 16777824
    Int_2_Loc 07FFFF60 = 16777824
    Int_3_Loc 07FFFF5C = 134217624
    Ch_Index  'C'
    Enum_Loc  07FFFF58 = <UNKNOWN: 255>
    Str_1_Loc 07FFFF38 = "\xFF\xFF\xFF\xFFx\x1E"
    Str_2_Loc 07FFFF18 = ""
    Run_Index 07FFFF64 = 16827048
    Number_Of_Runs100000
    n          <not alive>
  02. <not a source procedure. Address is 01001DF0>
```

The program was halted in `Proc_1` at line 309. The output shows that `Proc_1` was called from `main` line 170, and `main` was called by unnamed code at address `0x01001DF0`, which is part of the C runtime library.

Because `main` is called from the C runtime library, no source and no debug information is available for the procedure that called `main`, so `EXPAND` reports the pc address from which the call to `main` is made.

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *JOURNAL* on page 2-184

- *PRINTVALUE* on page 2-229
- *WHERE* on page 2-348.

2.3.58 FAILINC

The `FAILINC` command causes an abnormal exit from processing an include file.

Syntax

```
FAILINC  "string"
```

where:

`string`	A string to display that explains the reason for aborting the include file.

Description

The `FAILINC` command enables you to abort processing an include file. You might do this when checks of the target or debugger environment have failed to find resources the include file requires.

Use the string parameter to explain the abort.

Example

The following example shows how to use the `FAILINC` command in a macro:

```
if ( *((char*)(0xffe00)) != 0 )
  $failinc "Peripheral not initialized. Aborting$";
```

The following example shows how to use the `FAILINC` command in an include file:

```
jump nofail,( *((char*)(0xffe00)) == 0 )
failinc "Peripheral not initialized. Aborting"
:nofail
```

These two examples test a memory address, expecting to read a `0` from some peripheral register. If it does not read `0`, it aborts include file processing.

See also

The following commands provide similar or related functionality:

- *ERROR* on page 2-152
- *INCLUDE* on page 2-181
- *JUMP* on page 2-186.

2.3.59 FILL

The `FILL` command fills a memory block with values.

Syntax

`FILL [{/B|/H|/W|/8|/16|/32}] addressrange ={expression | expressionlist}`

where:

`/B, , /8`	Sets the fill size to byte (8 bits). If the processor naturally addresses bytes (for example, ARM7TDMI) then this is the default setting.
`/H, , /16`	Sets the fill size to halfword (16 bits).
`/W, , /32`	Sets the fill size to word (32 bits).
`addressrange`	Specifies the range of addresses that identify the memory contents to be filled with the pattern. The start and the end of the range is included in the range. For example a byte fill from `0x400..0x500` writes to `0x400` and to `0x500`. See *Specifying address ranges* on page 2-2 for more details on address ranges.

`expression` Specifies the pattern used to fill memory. The expression can be:

- a decimal or hexadecimal number
- a debugger expression, for example a math calculation
- a string enclosed in quotation marks.

If you use a quoted string:

- each character of the string is treated as a byte value in an *expressionlist*
- no C-style zero terminator byte is written to memory.

`expressionlist`

Specifies the pattern used to fill memory. An *expressionlist* is a sequence of values separated by commas, for example:

```
0x20,0x40,0x20
```

Note

All expressions in an expression string are padded or truncated to the size specified by the size qualifiers if they do not fit the specified size evenly. This also applies to each character of a string.

Description

The `FILL` command fills a memory block with values obtained from evaluating an expression or list of expressions. The size qualifier is used to determine the size of each element of *`expressionlist`*.

If the number of values in *`expressionlist`* is less than the number of bytes in the specified address range, the debugger repeatedly writes the list to memory until all of the designated memory locations are filled.

If more values than can be contained in the specified address range are given, the last repetition is completed before the process stops, so up to (`length(expressionlist)-1`) bytes, halfwords or words might be written beyond the range end address.

If you specify an address range with equal start and end addresses, the memory at that address is modified. If an expression is not specified, the debugger acts as if `=0` had been specified as the expression.

The `FILL` command runs synchronously unless background access to target memory is supported. Use the `WAIT` command to force it to run synchronously.

Examples

The following examples show how to use `FILL`:

`fill 0x1000..0x1004="hello"`

Writes `hello` in the locations `0x1000`...`0x1004`.

`fill 0x1000..0x1001="hello"`

Writes `hello` in the locations `0x1000`...`0x1004`.

`fill 0x1000..0x1013`

Writes as bytes the value `0` to locations `0x1000`...`0x1013`.

`fill /h 0x1000..0x1014`

Writes the 16-bit value `0` to locations `0x1000`...`0x1014`.

`fill 0x1000..0x1013="hello"`

Writes `hellohellohellohello` in the locations `0x1000`...`0x1013`.

`fill /w 0x2032..0x2053=0xDEADC0DE`

For a little-endian memory system, writes `0xDE` to `0x2032`, `0xC0` to `0x2033`, `0xAD` to `0x2034`, `0xDE` to `0x2035` and on to: `0xDE` to `0x2052`, `0xC0` to `0x2053`, `0xAD` to `0x2054`, and `0xDE` to `0x2055`.

```
fill 0x3000..0x4756 =0xEA000000/2
```

Writes `0x00` to `0x3000..0x4576`. The value of `0xEA000000/2` is calculated as `0x75000000`. Because fill defaults to a byte expression width, this is then truncated to `0x00` and written.

```
fill /32 0x3000..0x4756 =0xEA000000
```

Writes 1373 ARM processor `NOP` instructions to memory, changing locations `0x3000..0x4578`, and so writing 2 bytes more than the specified range.

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *MEMWINDOW* on page 2-201
- *SETMEM* on page 2-257.

2.3.60 FLASH

The FLASH command enables you to write, verify, or erase Flash blocks.

Syntax

```
flash [,qualifier...] [={addressrange | address, ...}]
```

where:

qualifier If specified, must be one or more of the following:

`cancel` Discard the patched or downloaded changes.

`erase` Erase the specified blocks. This normally sets every byte in the block to `0xFF` or `0x00`, depending on the type of Flash memory used. You can identify the Flash block using a handle, an address or an address range.

`write` Write data to the specified blocks of Flash memory. If you do not specify a block or address, then the write begins at the start of the first block.

`verify` If you specify this qualifier the data written to the Flash blocks is verified against the data source.

`handle=`*`blocknum,...`*

Identifies one or more Flash blocks to be operated on. These are the numbers on the left in the Open Flash Blocks list of the Flash Memory Control dialog. See the *RealView Developer Kit v2.2 Debugger User Guide* for details.

Note

Use the FLASH command with no arguments to view the Flash blocks.

Do not use an address range or address with this qualifier.

`useorig` This qualifier specifies that the original contents of the memory is used wherever it is not explicitly modified.

`scratch` This qualifier specifies that the original contents of the target memory buffer is not saved first. This might save you some time if the buffer is large.

By default the target memory buffer is saved first, and restored afterwards.

`addressrange`	Multiple Flash blocks can be specified using a range of addresses. The start and the end of the range is included in the range. For example, `0x24000000..0x24FFFFFF` specifies the blocks in the address range `0x24000000` to `0x24FFFFFF`. See *Specifying address ranges* on page 2-2 for more details. If you use the `handle` qualifier, then do not specify an address range.
`address`	The Flash block can be specified by address. If you use the `handle` qualifier, then do not specify an address.

Note

If you use the `address` or `addressrange` qualifiers, then those blocks that are touched by the `address` or `addressrange` are written to completely. For example, if you specify an address range that starts at block zero, and finishes part way into block three, then the whole of blocks one, two and three are written to.

Description

This command is used to manage Flash memory. This command enables you to:

- write and verify Flash blocks, which require that the blocks be opened first
- erase Flash blocks, which does not require the blocks to be open.

The Flash block is specified by `address`. You cannot program more than one target device at a time.

If this command is used with no arguments, it reports the currently open blocks.

Examples

This example shows information for the currently loaded Flash image:

```
> flash
Flash opened on ARM920T_0:ARM-ARM-NW for 'Intel DT28F320S3 2Mx16 x2 x4' at
'0x24000000'
  Block 0: +0x0000..+2644
```

This example erases the Flash in the range `0x24000000` to `0x24FFFFFF` inclusive.

```
flash,erase =0x24000000..0x24FFFFFF
```

See also

The following command provides similar or related functionality:

- *MEMMAP* on page 2-197.

2.3.61 FOPEN

The `FOPEN` command opens a file and assigns to it a specified file number.

Syntax

```
FOPEN [/A] [/R] fileid, filename
```

where:

`/A`	Appends new data to an existing file. You cannot read or write the existing information, and the existing information is retained.
`/R`	Opens a file as read-only. You must use this qualifier if you want to read the file with the `fgetc()` macro (see *fgetc* on page 3-12).
`fileid`	Specifies the identity of the file to be opened. This must be a user-defined *fileid*. See *Window and file numbers* on page 1-5 for details.
`filename`	Specifies the file being opened. Quotation marks are optional, but see *Rules for specifying filenames* on page 2-167 for details on how to specify filenames that include a path.

Description

This command enables you to read or write a file on the host filesystem by associating it with a RealView Debugger custom file number. However, `FOPEN` does not create a GUI window, and output is only sent to the file. If you require the output in a window, use the `VOPEN` command on page 2-343.

The file is opened for writes only by default, but you can specify append or read-only modes instead. You write to the file using the `FPRINTF` command, the `fputc` or `fwrite` macros, or by redirecting output from those commands that accept the `;` specifier. You read the file using either the `fgetc` or `fread` macros. You close the file using the `VCLOSE` command.

——— **Note** ———

Be aware of the following:

- The `FOPEN` command runs asynchronously unless it is used in a macro.
- If you open a new or existing file for writing, no data is written to the file until the output buffer is flushed. This happens when you close the file with the `VCLOSE` command, but it might happen at other times because the buffered I/O behavior is the same as in C.

- If you specify a filename with a path that does not exist, then an error message is displayed. RealView Debugger does not create the non-existent path.

Rules for specifying filenames

Follow these rules when specifying a filename:

- If the filename consists of only alphanumeric characters, slashes, or a period, but the filename does not start with a slash, then you do not have to use quotes. For example, `includes/file`.
- Filenames with a leading slash must be in double quotes, for example `"/file"`.
- Filenames containing a backslash must be in single quotes. For example `'\file'` or `'c:\myfiles\file'`.

  Alternatively, you can escape each backslash and use double quotes. For example, `"c:\\myfiles\\file"`.
- You can use environment variables to specify paths to a file. For example, if `PATHROOT=C:\MYFILES` and `PATHTEST=TEST1`:

  `'$PATHROOT\$PATHTEST\test1.c'`

  You can include:

  — the filename as part of the second environment variable, and then specify `'$PATHROOT\$PATHTEST'`.

  — the path separator in the environment variable, and then specify `'$PATHROOT$PATHTEST'`.

Examples

The following examples show how to use `FOPEN`:

```
fopen 50, 'c:\temp\file.txt'
fprintf 50, "Start of function\n"
```

Open a file and write some text to it.

```
fopen /r 50, 'c:\temp\file.txt'
ce fgetc(50)
```

Open a file and read the first character of the file.

See also

The following commands provide similar or related functionality:

- *FPRINTF* on page 2-169
- *VCLOSE* on page 2-337
- *VMACRO* on page 2-341
- *VOPEN* on page 2-343
- *WINDOW* on page 2-350.

The following macros provide similar or related functionality:

- *fgetc* on page 3-12
- *fopen* on page 3-15
- *fputc* on page 3-17
- *fread* on page 3-19
- *fwrite* on page 3-21.

2.3.62 FPRINTF

The `FPRINTF` command displays formatted text to a specified file or window.

Syntax

`FPRINTF {windowid | fileid}, "format_string" [,argument...]`

where:

`windowid | fileid`

Specifies a window or file where the formatted text is to be sent. This must be a user-defined *`windowid`* or *`fileid`*. See *Window and file numbers* on page 1-5 for details.

`format_string`

Is a format specification conforming to C/C++ rules with extensions. It might be a text message, or it can describe how one or more arguments are to be presented. See *Format string syntax* for details.

`argument`	The value or values to be written.

Description

The command is similar to the C run-time `fprintf` function. You select the *`windowid`* or *`fileid`* to use from the range `50..1024`. For output to a file, the file must be opened using the `FOPEN` command. For output to a user window, the window must be opened using the `VOPEN` command.

Format string syntax

The text in *`format_string`* is defines what is displayed. If there are no `%` characters in the string, the text is written out and any other arguments to `FPRINTF` are ignored. The `%` symbol is used to indicate the start of an argument conversion specification. The syntax of the specification is:

`%[flag][fieldwidth][precision][lenmod]convspec`

where:

`flag`	An optional conversion modification flag `-`. If specified, the result is left-justified within the field width. If not specified, the result is right-justified.
`fieldwidth`	An optional minimum field width specified in decimal.

precision An optional precision specified in decimal, with a preceding `.` (period character) to identify it.

lenmod An optional argument length specifier:

h	a 16-bit value
l	a 32-bit value
ll	a 64-bit value

convspec The possible conversion specifier characters, `<convspec>`, are:

% A literal `%` character.

m The mnemonic for the processor instruction in memory pointed to by the argument. The expansion includes a newline character. The information that is printed includes:

- the memory address in hexadecimal
- the memory contents in hexadecimal
- the instruction mnemonic and arguments
- an ASCII representation of the memory contents, if printable.

H A line from the current source file, where the argument is the line number.

h A line from the current source file, where the argument is the source line address (as opposed to a target memory address).

d, i, or u An integer argument printed in decimal. **`d`** and **`i`** are equivalent, and indicate a signed integer. **`u`** is used for unsigned integers.

x or X An integer argument printed in unsigned hexadecimal. **`x`** indicates that the letters `a` to `f` are used for the extra digits, and **`X`** indicates that the letters `A` to `F` are used.

c A single character argument.

s A string argument. The string itself can be stored on the host or on the target.

p A pointer argument. The value of the pointer is printed in hexadecimal.

e, E, f, g, or G

A floating point argument, printed in scientific notation, fixed point notation, or the shorter of the two. The capital letter forms use a capital `E` in scientific notation rather than an `e`.

Output is formatted beginning at the left of the format string and is copied to the screen. If you are using the GUI, then the string is copied to the Output pane. Whenever a conversion specification is encountered, the next argument is converted according to the specification, and the result is copied to the screen.

Rules

The following rules apply to the use of the `FPRINTF` command:

- `FPRINTF` runs synchronously
- *`windowid`* must identify a user-defined window that you have previously opened with the `VOPEN` command
- *`fileid`* must identify a file that you have previously opened in write mode, for example:

  ```
  FOPEN 100, "c:\myfiles\file.txt"
  ```
- if there are too many arguments, some of them are not printed
- if there are too few arguments (that is, there are more conversion specifiers in the format string than there are arguments after the format string), the string `<invalid value>` is output instead
- if the argument type does not correspond to its conversion field specification, arguments are converted incorrectly.

Example

The following examples show how to use `FPRINTF`:

```
fprintf 50,"Syntax error\n"
```

Write the string `Syntax error` to the window or file.

```
fprintf 50, "Execution time: %d seconds\n", tend-tstart
```

Print the result of the calculation to the window or file, in the format:

```
Execution time: 20 seconds
```

```
fprintf 50, "Value is %d\n"
```

Print the following to the window or file:

```
Value=<invalid value>
```

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *FOPEN* on page 2-166
- *PRINTF* on page 2-220
- *PRINTVALUE* on page 2-229
- *VOPEN* on page 2-343
- *VCLOSE* on page 2-337.

The following macros provide similar or related functionality:

- *fopen* on page 3-15
- *fputc* on page 3-17
- *fread* on page 3-19.

2.3.63 GO

The `GO` command executes the target program starting from the current PC or from a specified address.

Syntax

```
GO [=start_address[,]] [ {temp_break [%%passcount][,] }... [;macro_call]]
```

where:

start_address	Specifies an address at which execution is to begin.
temp_break	Acts as a temporary instruction breakpoint, which is automatically cleared when program execution is suspended.
passcount	Specifies the number of times the *temp_break* address is executed before the command actually halts.
macro_name	Invokes a macro if a temporary break occurs. The macro return value determines whether execution continues or not. If there is an attached macro, execution continues when the macro returns a non-zero value. If the macro returns a zero, execution halts.

Description

This command executes the target program starting from the current PC or from a specified address. The command also causes program execution to resume after it has been suspended. Execution continues until a permanent or temporary breakpoint, an error, or a halt instruction is encountered. You can also click **Stop** to halt execution.

RealView Debugger continues to accept commands after `GO` has been entered. Commands that cannot be completed while the target is running (synchronous commands) are delayed until the target is next stopped. You can stop the target by clicking **Stop**. For more information about the limitations the target vehicle imposes while the target is running, see your target documentation.

You can specify a temporary instruction breakpoint with the `GO` command, providing similar functionality to the **Go to Cursor** GUI command. The temporary breakpoint is removed as soon as the target next stops, whether the breakpoint was hit or not. You can also associate a macro to be run that can also determine whether the target remains stopped at the breakpoint.

The `GO` command runs synchronously.

If you are working with RTOS images, and the current connection is running in RSD mode, then the GO command starts the current thread. See the chapters that describe RTOS support and working with multiple target connections in the *RealView Developer Kit v2.2 Extensions User Guide*.

Note

When specifying a start address you must be careful to make sure that the processor stack has been set up and remains balanced.

Examples

The following examples show how to use `GO`:

`GO`	Start or resume executing the target program from the current PC.
`GO @1`	Resume executing the target program from the current PC, stopping when the current function returns to its caller.
`GO write_io; until (x==2)`	Resume executing the target program from the current PC, and stop when `x` has the value 2.

See also

The following commands provide similar or related functionality:

- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *HALT* on page 2-177
- *GOSTEP* on page 2-175
- *RUN* on page 2-250
- *STEPINSTR* on page 2-273
- *STEPLINE* on page 2-275
- *STOP* on page 2-283.

2.3.64 GOSTEP

The `GOSTEP` command single-steps through the program, invoking a named macro at every step.

Syntax

```
GOSTEP macro_name
```

where:

macro_name Specifies the name of the macro that is invoked after each instruction.

The macro return value determines whether execution continues or not. Execution continues when the macro returns a non-zero value.

Description

The `GOSTEP` command single-steps through the program, invoking a named macro at every step. Execution starts at the current PC, and continues until you click **Stop** to halt execution, the macro returns zero, or a breakpoint is hit. Single-stepping is by source line for high-level source code and by processor instruction for assembly language code.

The `GOSTEP` command runs synchronously.

Note

- Using the command significantly slows target execution speed.
- Using the command might cause target program execution errors because of timing issues.

Example

The following examples show how to use `GOSTEP`:

`GOSTEP checkvariable`

Start or resume executing the target program from the current PC. At each step, invoke a macro called `checkvariable`. A step is an instruction or a statement, depending on the source display `MODE`.

`GOSTEP until (y>100)`

Resume executing the target program, stopping when the program variable `y` exceeds `100`. `until` is a predefined macro.

See also

The following commands provide similar or related functionality:

- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *HALT* on page 2-177
- *GO* on page 2-173
- *MODE* on page 2-203
- *RUN* on page 2-250
- *STEPINSTR* on page 2-273
- *STEPLINE* on page 2-275
- *STOP* on page 2-283.

2.3.65 HALT

The `HALT` command stops target program execution. In RSD mode, this stops the current thread.

Syntax

```
HALT
```

Description

If no OS support is enabled, or HSD support is enabled, this stops the processor.

If RSD is enabled, this stops the currently running thread, or the thread attached to the Code window.

If you are working with RTOS images, see the chapters that describe RTOS support and working with multiple target connections in the *RealView Developer Kit v2.2 Extensions User Guide*.

See also

The following commands provide similar or related functionality:

- *AOS_resource-list* on page 2-32
- *BREAKINSTRUCTION* on page 2-61
- *DOS_resource-list* on page 2-129
- *GO* on page 2-173
- *GOSTEP* on page 2-175
- *OSCTRL* on page 2-214
- *RUN* on page 2-250
- *STEPINSTR* on page 2-273
- *STOP* on page 2-283.

2.3.66 HELP

The `HELP` command displays RealView Debugger online help. To do this type:

```
HELP
```

The topic **Welcome to RealView Debugger Help** includes more information about using online help in RealView Debugger.

Note

This command has no effect in the headless debugger. Use the `DHELP` or `DCOMMANDS` instead. See *DHELP* on page 2-119 and *DCOMMANDS* on page 2-107.

2.3.67 HOST

The `HOST` command enables you to run a command on your host operating system.

Syntax

`HOST` *`command`*

where:

`command`	The command that you want to run on your host operating system. This is a DOS command on a Windows system.

Description

The `HOST` command enables you to run a command on your host operating system (that is, Windows).

Note

You cannot use the `HOST` command to change the current directory pointed to by RealView Debugger. For example, `host cd "c:\my sources"` has no effect.

Examples

The following examples show how to use `HOST` on Windows system:

`host dir "c:\my sources"`	Lists the contents of directory `c:\my sources`. This must be in quotes because there is a space in the path name.
`host cd`	Displays the current directory pointed to by RealView Debugger.

2.3.68 HWRESET

`HWRESET` is an alias of `EMURESET` (see page 2-148).

2.3.69 INCLUDE

The `INCLUDE` command executes commands stored in the specified file.

Syntax

`INCLUDE [/S] filename`

where:

`/S`	Stops the commands in the include file being echoed to the display. However, the commands are still added to the command history list.
`filename`	Specifies the command file to be read. Quotation marks are optional, but see *Rules for specifying filenames* on page 2-167 for details on how to specify filenames that include a path.

Description

The `INCLUDE` command executes a group of commands stored in the specified file as though they were entered from the keyboard. Commands in the file are executed until the end of the file is reached or an error occurs. If an error occurs, the debugger behaves as specified by the `ERROR` command. If a filename extension is not specified, the debugger automatically appends the extension `.inc`.

Note

If you want to include a batch file when a target is running, you must first enter the `wait=off` command, then include the batch file:

```
> wait=off
> include myfile.inc
```

Your batch file can still include the `wait=on` command, if required.

The `INCLUDE` command is normally used to perform repetitive or complex initializations, such as:

- loading and running programs, setting up breakpoints and initial variable definitions
- creating debugger aliases and macros, perhaps for use in later debugging

  Note

  The `DEFINE` command, used to create macros, can only be used in an `INCLUDE` file.

- running test suites.

You can configure the debugger to load a given include file automatically when a target connection is made using the `Commands` setting of the `Advanced_Information` block for your target.

You can also run script files using the `-inc` argument to RealView Debugger itself.

The `INCLUDE` command runs asynchronously unless in a macro.

Example

The following example shows how to use `INCLUDE`:

```
INCLUDE "startup.inc"
```

Read the file `startup.inc` in the current directory and interpret the contents as RealView Debugger commands. The file `startup.inc` might contain:

```
; startup.inc 12/12/00
; Author: J.Doe
;
alias sf*ile =dtfile ;99
alias dub =dump /b
vopen 99
```

See also

The following commands provide similar or related functionality:

- *ALIAS* on page 2-25
- *DEFINE* on page 2-110
- *ERROR* on page 2-152
- *FAILINC* on page 2-160
- *JUMP* on page 2-186
- *MACRO* on page 2-195
- *WAIT* on page 2-346.

2.3.70 INTRPT

The `INTRPT` command interrupts the execution of commands.

Syntax

```
INTRPT
```

Description

The `INTRPT` command enables you to interrupt an asynchronous command that the target is still executing. Commands are held in a queue for execution when the target stops. This is called *pending* the command.

Use the `CANCEL` command to clear pending commands from the list, to stop them being executed.

You cannot use this command to halt target execution. Use `HALT` to do this.

——— **Note** ———

Synchronous commands can only be run when target program execution has stopped.

Asynchronous commands can be run at all times.

————————

See also

The following commands provide similar or related functionality:

- *CANCEL* on page 2-89
- *HALT* on page 2-177
- *WAIT* on page 2-346.

2.3.71 JOURNAL

The `JOURNAL` command controls the logging of commands and output.

Syntax

```
JOURNAL [/A] [{OFF | ON="filename"}]
```

where:

`/A`	Appends information to an existing file.
`OFF`	Closes the journal file and stops collecting information. This is the default setting.
`ON`	Starts writing information to the journal file.
`filename`	Specifies the journal filename. If you do not specify a filename extension, the extension `.jou` is used. Quotation marks are optional, but see *Rules for specifying filenames* on page 2-167 for details on how to specify filenames that include a path.

Description

The `JOURNAL` command starts or stops saving, in a specified file:

- the commands that you enter
- any output that is generated by a command
- error messages
- text specifically sent to the journal file.

If you are using the GUI, then the log file contains the same information that is displayed in the **Cmd** tab of the Output pane.

Note

If the specified file exists and you do not specify the `/A` parameter, the existing contents of the file are overwritten and lost.

The `JOURNAL` command runs asynchronously unless it is in a macro.

Example

The following examples show how to use `JOURNAL`:

`JOURNAL ON='c:\temp\log.txt'`

Start logging output to the file `c:\temp\log.txt`, overwriting any existing file of that name.

`JOURNAL /A ON="log"`

Start logging output to the file `log.jou` in the current directory of the debugger, appending the new log text to the file if it already exists.

`JOURNAL OFF`

Stop logging output.

See also

The following commands provide similar or related functionality:

- *LOG* on page 2-193
- *STDIOLOG* on page 2-271
- *VOPEN* on page 2-343.

2.3.72 JUMP

The JUMP command goes to a label in an include file.

Syntax

<u>JUMP</u> *label* [,*condition*]

where:

label	Is the string that identifies the target line in the include file to which you want control to jump. The first character of the target label must be a colon :, and it must be followed by a label string.
condition	Is an optional expression that can be evaluated as True or False. The jump to the specified label takes place only if the condition is True, otherwise control passes to the next command in the include file.

Description

The JUMP command can only be used in an include file. If you specify a condition, then the jump takes place only if the condition is True. Otherwise control passes to the next line in the include file.

You cannot use the JUMP command inside a macro, nor place a target label inside a macro. However, you can provide similar functionality by using the **if**, **for**, **while** and **do-while** flow control constructs in macros (see Chapter 4 *RealView Debugger Keywords*).

Example

The following fragment of an include file shows the use of labels and jumps:

```
initialize
:retry
jump skip_setup,x==1 // variable x is 1 when setup is complete
some_commands
jump retry           // keep trying to initialize
:skip_setup
```

See also

The following commands provide similar or related functionality:

- DEFINE on page 2-110
- FAILINC on page 2-160

2.3.73 LIST

The `LIST` command displays source code in the Code window.

Syntax

`LIST [{#line_number | function_name | @stack_level}]`

where:

line_number	Specifies the number of the first line to be displayed.
function_name	Specifies a function that is to have its source code displayed.
@stack_level	Displays the line that is returned to after the specified nesting level. For example, `@1` represents the instruction after the call to the current procedure.

Description

The `LIST` command displays the source code in the Code window beginning at the specified line number, stack level, or function name.

You can qualify line number or procedure names by preceding them with a module name. If you do not specify a parameter for the `LIST` command, the line pointed to by the PC is displayed.

The `LIST` command runs asynchronously unless in a macro.

Example

The following examples show how to use `LIST`:

`list`	List the text of the current source file from the current PC location, if that refers to a source file with debugging information.
`list #44`	List the text of the current source file from line 44.
`list @1`	List the text of the source file containing the call to the current procedure, starting from the statement after the call.

See also

The following commands provide similar or related functionality:

- *CONTEXT* on page 2-102

- *DISASSEMBLE* on page 2-122
- *DOWN* on page 2-131
- *EXPAND* on page 2-157
- *SCOPE* on page 2-252
- *UP* on page 2-335
- *WHERE* on page 2-348.

2.3.74 LOAD

The `LOAD` command loads the specified executable file into the debugger.

Syntax

```
LOAD [/A] [/C] [/NI] [{/NP|/SP}] [/NS] [/PD] [/R] absolute_filename[,root]
[;section [,section]...] [;arg1 ...] [&base_address]
```

where:

`/A` This loads and appends another executable image without deleting any existing one. If the new image file overlaps the addresses of the existing object modules, the load terminates and displays an error message. If you want to replace the current image with a new one, use `/R`.

This option might be the default option if you are running an operating system extension to RealView Debugger. For more information, consult the manual provided with the extension.

`/C` Converts all symbols to lowercase as they are read by the absolute file reader.

`/NI` Loads only the symbol table. Overlap of addresses is checked if `/A` is also used. Does not load the program image code or the data.

`/NP` Prevents the command changing the value of the PC.

`/NS` Prevents the command loading debug information into the symbol table. Only the program image is loaded. No check for overlapping addresses is made. The `/NS` option can be used to reload the current program image without affecting the symbol table.

`/PD` Pop dialog. Display a dialog for errors and warnings, rather than dumping them to the log.

`/R` Replaces the existing program with the program being loaded.

`/SP` Sets the PC to the start address specified in the object module. This is the default behavior when symbols are loaded, the image file specifies an entry address and the `/A` flag is not also specified.

`absolute_filename`

Specifies the name of the absolute object file to be loaded. Quotation marks are optional, but see *Rules for specifying filenames* on page 2-167 for details on how to specify filenames that include a path. Also, see *Rules* on page 2-191.

`root`	Specifies the root associated with the symbols in the program being loaded. The default root is the filename without an extension. See *Rules* on page 2-191 for details on how to specify a root.
`section`	Lists sections to load when an image is being loaded. The default is to load all sections. This option is commonly used to reload the initialized data area when starting a program. The section names that are available for a specific image can be listed using the ARM development tools command `fromelf` or the GNU development tools command `objdump`. See *Rules* on page 2-191 for details on how to specify sections.
`args`	Specifies an optional, space-separated, list of arguments to the image. **— Note —** You can also specify arguments using the `ARGUMENTS` command. For example, you can might want to modify the arguments without unloading the image. The case of arguments is preserved. See *Rules* on page 2-191 for details on how to specify arguments.
`base_address`	Specifies an address offset to be added to all sections when computing the load addresses. For this option to work correctly, the program must have been compiled with *Position-Independent Code* (PIC) and *Position-Independent Data* (PID).

Description

The `LOAD` command loads the specified executable file into the debug target. The file specified must be a format supported by the RealView Debugger.

To reset the initialized values of program variables after entering a `RESET` or a `RESTART` command, you must reload your program using the `LOAD` command. The `RELOAD` command checks the file date to determine whether program symbols have changed and therefore whether they must be reloaded.

If a load is performed that includes the symbol table, any breakpoints or macros referring to symbols in the previous root are invalidated.

The `LOAD` command runs synchronously.

Rules

Follow these rules when using `LOAD`:

- *`absolute_filename`*, *`root`*, *`section`*, and *`args`* must all be placed in the same set of quotes. For example, on Windows:

  ```
  load /pd/r 'c:\source\demofile.axf ;ER_RO,ER_ZI ;12345' &0x8A00
  ```

- If you want to specify arguments, but not a section, you must specify an empty section. All sections are loaded in this case. For example:

  ```
  load /pd/r 'c:\source\myfile.axf;;arg1 arg2 arg3'
  ```

- Where an argument includes spaces, additional quotes must be used. Use single quotes around arguments if the outer quotes are double quotes. Use double quotes around arguments if outer quotes are single quotes. For example:

  ```
  load /pd/r "myimage.axf ;;12345 'Argument Two'" &0x8A00
  ```

- *`base_address`* must be placed outside the quotes, and must be the last parameter specified.

Examples

The following examples show how to use `LOAD`:

```
load 'c:\source\myfile.axf'
```

Load the executable file `myfile.axf` to the target.

```
load /ni/sp 'c:\source\rtos.axf'
```

Load the symbol table for an image `rtos.axf` that is also in target ROM, setting the PC to the program start address so that a subsequent `GO` runs the program.

```
load /np 'c:\source\mp3.axf'
```

Load the executable library `mp3.axf` onto the target so that the preloaded executable can use it. The PC is not modified. Symbol table entries in `mp3.axf` are added to the existing symbol table.

Note

Ensure that executables you load in this way occupy distinct memory regions. No relocation is performed by RealView Debugger unless you specify a base offset.

```
load /pd/r 'c:\source\demofile.axf ;ER_RO,ER_ZI' &0x8A00
```

Load the executable file `demofile.axf` to the default target. Specify an offset added to all sections to compute the load addresses. Load only the specified sections `ER_RO` and `ER_ZI`.

```
load /pd/r 'c:\source\myfile.axf;;arg1 arg2 arg3'
```

Load the executable file `myfile.axf` to the default target using an arguments list. An empty *section* list is given so all sections are loaded.

See also

The following commands provide similar or related functionality:

- *ADDFILE* on page 2-23
- *ARGUMENTS* on page 2-34
- *DTFILE* on page 2-137
- *GO* on page 2-173
- *RELOAD* on page 2-239
- *RESET* on page 2-241
- *RESTART* on page 2-245
- *RUN* on page 2-250
- *UNLOAD* on page 2-333.

2.3.75 LOG

The `LOG` command records user input and places it in a specified file.

Syntax

`LOG [/A] [{OFF | ON="`*`filename`*`"}]`

where:

`/A`	Specifies that new records are to be added to any that already exist in the specified file.
`OFF`	Closes the log file and stops collecting information. This is the default.
`ON`	Starts writing information to the log file.
`filename`	Specifies the name of the log file. Quotation marks are optional, but see *Rules for specifying filenames* on page 2-167 for details on how to specify filenames that include a path.

Description

This command records user input and places it in a specified file. Commands that are issued but not successfully completed are written to the log file as comments along with the associated error codes. All successful commands are written to the log file, so the file can be used as an include file.

Note

If you want to use the log file as an include file, first remove the `log` command that appears at the start of the file.

If the specified file exists and you do not specify the `/A` parameter, the existing contents of the file are overwritten and lost. A window number (28) is associated with the log file so that text can be written to it using `FPRINTF`. See *Window and file numbers* on page 1-5 for details.

Using `LOG` with no parameters shows the current log file, if any. User input is recorded in the log file until the `LOG OFF` command is issued.

The `LOG` command runs asynchronously unless in a macro.

Example

The following examples show how to use `LOG`:

`LOG ON='c:\temp\log.txt'`	Start logging output to the file `c:\temp\log.txt`, overwriting any existing file of that name.
`LOG /A ON="log"`	Start logging output to the file `log.log` in the current directory of the debugger, appending the new log text to the file if it already exists.
`LOG OFF`	Stop logging output.

See also

The following commands provide similar or related functionality:

- *JOURNAL* on page 2-184
- *STDIOLOG* on page 2-271
- *VOPEN* on page 2-343.

2.3.76 MACRO

The `MACRO` command enables you to run a predefined or user-defined macro.

Syntax

`MACRO` *`macroname`*`(`*`parameters`*`...)`

where:

`macroname`	Specifies that name of the macro.
`parameters`	The actual values of parameters required by the macro.

Description

The `MACRO` command runs a macro. You can run macros in these ways:

- as part of the expression in a `CE` command
- as the argument to the `MACRO` command
- as a command on its own.

The `CE` command enables you to see the result of the macro, as set with the `RETURN` statement. If the macro does not explicitly return information, or you do not have to know the return value, you can use the macro name as a command. However, in this case the macro is only run if the name does not match any other debugger command or any alias defined with `ALIAS`. You can therefore use the `MACRO` command to ensure that the command that is run is the macro, and not a debugger command or an alias.

Note

It is recommended that, if you call macros in an include file and they do not return a value, that you use `MACRO` to make the call. This ensures that the future operation of the include file is not changed if new commands are added to the debugger, for example using `ALIAS`.

Macros can also be invoked as actions associated with:

- a window, for example `VMACRO`
- a breakpoint, for example `BREAKEXECUTION`
- deferred commands, for example `BGLOBAL`.

Note

Macros that are not directly invoked from the command line cannot execute `GO`, or `GOSTEP`, or any of the stepping commands, for example `STEPINSTR`.

Example

The following example shows how to use `MACRO`:

`macro fgetc(50)`

Read a character from the file associated with the file number `50` and throw it away, with the side effect of advancing the file pointer to the next character.

See also

The following commands provide similar or related functionality:

- *ALIAS* on page 2-25
- *CEXPRESSION* on page 2-91
- *DEFINE* on page 2-110
- *INCLUDE* on page 2-181
- *PRINTSYMBOLS* on page 2-224.
- *SHOW* on page 2-269
- *VMACRO* on page 2-341.

2.3.77 MEMMAP

The `MEMMAP` command enables you to define and control memory mapping.

Syntax

`MEMMAP [,`*`qualifier`*`...] [={`*`address`*`|`*`address_range`*`}]`

where:

`qualifier` One of the following:

`enable` Turns on memory mapping control. This is the default.

The debugger only accesses the target memory in regions that are defined in the map, and uses the access method to determine the operations that are permitted.

`disable` Turns off memory mapping control. The debugger assumes that all memory is RAM.

`delete` Deletes memory map entries:

- if you supply a memory map entry start address in *`address`*, delete that entry
- if you supply no arguments, delete all memory maps.

`autosection` When loading an image, create memory mappings automatically from the sections of the image. This is default behavior.

`updateautomap` Update the memory map based on the information provided in the board file. This is automatically done when:

- the debugger starts up
- the target program stops
- the registers that control the map are changed by you.

This qualifier enables you to manually request a map update.

`define` Creates a new memory region using the address range in *`address`*. You can specify additional information about the region with the `type`, `access`, and `description` qualifiers.

`descr``iption:`*`text`*	Set the name of this memory map region to *`text`*. This is used to label the entry for your own reference.
`access:`*`text`*	Set the memory access type to *`text`*, which must be one of the predefined strings:

`RAM`	memory can be read and written with no specific provision.
`ROM`	memory can only be read.
`WOM`	memory can only be written.
`NOM`	there is no memory in this region.
`Flash`	there is Flash memory in this region. It can always be read, and it can be written as required using the Flash memory procedure if this is defined.
`Auto`	There is memory in this region but the type is inferred by the image that is loaded. Memory in regions not defined by the image are assumed to be absent (equivalent to `NOM`).
`Prompt`	There is memory in this region but you set the type by responding to a prompt when loading an image to it. The default is there is no memory.

`asize:`*`size`* The size of memory accesses, where *`size`* is one of:

1	1-byte accesses
2	2-byte accesses
4	4-byte accesses
8	8-byte accesses

`type:`*`text`*	Set the memory type to *`text`*, which must be one of the defined memory type strings for the processor architecture. For ARM processors, the only available type is `Any`.
`fme:`*`filename`*	For Flash memory, the Flash programming method file (`*.fme`) that is to be used. You must enter the full path and file name, enclosed in double quotes.

`address` The memory region, specified as a single address (`delete`).

address_range

The memory region, specified as an address range (`define`). The start and the end of the range is included in the range. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

The `MEMMAP` command enables you to define and control memory mapping. You can enable and disable mapping, and define new memory regions based on type and access rights. The list of valid access rights and types is defined by the vehicle and processor.

Note

Debugger internal handle numbers are not available to users to identify memory blocks. There is no command that lists out the memory maps with the map handle numbers.

Examples

The following examples show how to use `MEMMAP`:

```
memmap,define 0x10000..0x20000
```

Creates a memory map region from `0x10000` to `0x20000`, inclusive. The length of the region is `0x10001` bytes.

```
memmap,define 0x10000..+0x10000
```

Creates a memory map region from `0x10000` to `0x1FFFF`, inclusive. The length of the region is `0x10000` bytes.

```
mmap,def,access:Flash,type:Any,asize:4,descr:"Intel",fme:"C:\myflash\IntegratorA
P\flash_IntegratorAP.fme"=0x24000000..0x25FFFFFF
```

Define a Flash memory region called Intel, using 4-byte memory accesses, and using the Flash programming method file located in `C:\myflash\IntegratorAP\flash_IntegratorAP.fme`.

```
mmap,def,access:RAM,type:Any,description:"Data space"=0x0000..0x7FFF
```

Define a read/write memory region called `Data space` in the first 32KB of memory.

```
mmap,def,access:ROM,type:Any,descr:"Bootrom"=0x10000..+0xFFFF
```

Define the 64KB region starting at `0x10000` as a read-only region called `Bootrom`.

`mmap,delete =0x10000`	
	Delete the memory map entry that starts at `0x10000`, resetting the map for that area to the `Auto` map.
`mmap,delete`	Delete all memory map entries, resetting the map to the default `Auto` map over the whole address space.
`mmap,disable`	Disable memory mapping.

Alias

`MMAP` is an alias of `MEMMAP`.

See also

The following commands provide similar or related functionality:

- *DUMPMAP* on page 2-143
- *FLASH* on page 2-164
- *MEMWINDOW* on page 2-201
- *SETMEM* on page 2-257.

2.3.78 MEMWINDOW

The MEMWINDOW command sets the base address of the memory pane.

Note

This command is not available in the headless debugger.

Syntax

```
MEMWINDOW [{/B|/H|/W|/8|/16|/32}] address
```

where:

`/B, , /8`	Sets the display format to byte (8 bits). If the processor naturally addresses bytes (for example, ARM7TDMI) then this is the default setting.
`/H, , /16`	Sets the display format to halfword (16 bits).
`/W, , /32`	Sets the display format to word (32 bits).
`address`	The base address for the memory pane.

Description

The `MEMWINDOW` command sets the base address of the memory pane. You can specify the size of each printed value using the qualifiers. If you do not specify a size, the previous size is retained.

Example

The following example shows how to use `MEMWINDOW`:

```
memw /b 0x200
```

Display in the memory pane bytes from address `0x200`.

See also

The following command provides similar or related functionality:

- *SETMEM* on page 2-257.

2.3.79 MMAP

`MMAP` is an alias of `MEMMAP` (see page 2-197).

2.3.80 MODE

The `MODE` command switches the code window between disassembly and source view.

——— Note ———

This command has no effect in the headless debugger.

Syntax

```
MODE [{HIGHLEVEL | ASSEMBLY}]
```

where:

`HIGHLEVEL`	Set the code window to the source view.
`ASSEMBLY`	Set the code window to the disassembly view.

Description

The `MODE` command enables you to toggle between disassembly and source modes of the Code view, and along with this, the stepping mode of the `GOSTEP` command. Without an argument, the current mode is toggled. With an argument, the current view mode is set to the indicated mode.

See also

The following commands provide similar or related functionality:

- *CONTEXT* on page 2-102.
- *DISASSEMBLE* on page 2-122.
- *GOSTEP* on page 2-175
- *LIST* on page 2-187.

2.3.81 MONITOR

The `MONITOR` command adds the named variable to the list of monitored, or watched, variables, displayed in the Watch pane.

Note

This command is not available in the headless debugger.

Syntax

`MONITOR` *`variable_name`*

where:

`variable_name`	The name of a variable or expression in the current context, or a path name, using the `\\module\proc\variable` syntax, for a variable that you are monitoring.

Description

The `MONITOR` command adds a variable to the list of watched variables displayed in the Watch pane of the debugger. This list displays the values of each variable every time the debugger stops, for example at a breakpoint. If the variable is out of scope when the debugger stops, the value is printed as `Symbol not found without qualification`.

You can add pointer and structure variables to this list. If you do, the values of members and referenced variables can be displayed using the ⊞ icon next to the pointer name in the watch pane.

Note

- `MONITOR` is equivalent to `display`, found in some other debuggers.
- You can print the value of a variable using the `CEXPRESSION` or `PRINTVALUE` command.

Examples

The following examples show how to use `MONITOR`:

`monitor count`

Monitor the value of the variable count, displaying the value as an integer.

`moni this` Monitor the members of the current C++ class, through the C++ class pointer `this`.

`moni \\MAIN_1\ALLOC\maxalloc`

Monitor the global variable `maxalloc` from the file `main.c`.

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *CONTEXT* on page 2-102
- *DUMP* on page 2-141.
- *NOMONITOR* on page 2-208
- *PRINTVALUE* on page 2-229.

2.3.82 NAMETRANSLATE

The `NAMETRANSLATE` command manipulates the host/target name translation list.

Syntax

`NAMETRANSLATE [,qualifier] [=name-translation, ...]`

where:

qualifier If specified, must be one of the following:

`replace`	The current name translation list is to be replaced by the list specified in this command.
`delete`	The current name translation list is to be deleted.

If you do not supply a qualifier, the name translation list specified in this command is appended to the current name translation list.

name_translation

A list of name translations. The format is:

`"hname=tname"`

where `hname` is the filename on the host and `tname` is the filename on the target. Both `hname` and `tname` can be a comma-separated list.

Each translation must be enclosed in double quotes.

Description

The `NAMETRANSLATE` command extends, replaces, or deletes the name translation list. Name translation enables a host filename to be different from a target filename.

If you supply no arguments, the `NAMETRANSLATE` command prompts you to enter a name translation for the current board:

- If you enter this command in the **Cmd** tab of the RealView Debugger GUI, a prompt dialog is displayed and shows any existing translation.
- If you enter this command in the headless debugger, you can press Enter to show the current name translation.

If translating from host name to target name, the first target name in the list is used. If translating from target name to host name, the first host name in the list is used.

Examples

The following examples show how to use `NAMETRANSLATE`:

```
nametranslate "hostfile1=targetfile1","hostfile2=targetfile2"
```

```
nametranslate "hostfile1,hostfile2,hostfile3=targetfile1,targetfile2"
```

See also

The following commands provide similar or related functionality:

- *LOAD* on page 2-189
- *PATHTRANSLATE* on page 2-215.

2.3.83 NOMONITOR

The `NOMONITOR` command deletes variables from the Watch pane.

Note

This command is not available in the headless debugger.

Syntax

```
NOMONITOR [{linenum | linenum..linenum}]
```

where:

linenum	A line number or a line number range for the items to delete.

Description

This `NOMONITOR` command deletes variables added to the Watch pane by `MONITOR`, using a line number in the pane to identify the item to delete.

Line numbers start at 1 for the first line and increment by one for each top-level variable. A structure or array variable that has been expanded using the icon to the left of the variable name, ⊞, counts as only one line. If you reference a line that is not present, the command is ignored.

You can delete several consecutive elements from the Watch pane using a line number range, separating the first and last line numbers with a double-dot (`..`). If the end of a line range is not present, only the lines that are present are deleted.

If you do not specify a line number or line number range, all lines are deleted from the Watch pane.

Examples

The following examples show how to use `NOMONITOR`:

`nomonitor 2` Delete the variable on line 2 of the Watch pane.

`nomonitor 2..4`

Delete the variables on lines 2, 3, and 4 of the Watch pane.

See also

The following command provides similar or related functionality:

- *MONITOR* on page 2-204.

2.3.84 ONSTATE

The `ONSTATE` command executes the associated command when a particular event occurs.

Syntax

`ONSTATE [,`*`event`*`] [,timer] [,replace] [command]`

where:

`event`	Specifies the event to trigger on from the following list:
	`start` Execute the command immediately before program execution starts.
	`stop` Execute the command immediately after program execution stops.
	`starttimed` Execute the command immediately before program execution starts and at the specified interval thereafter until the program stops running. The target must support execution of commands on a running target.
	`tstart` An alias of `starttimed`.
	`stoptimed` Execute the command immediately after program execution stops and at the specified interval thereafter, until the debugger starts the program again or the target is disconnected. Specify the time interval using the `,timer` qualifier, with the interval in milliseconds.
	`tstop` An alias of `stoptimed`.
	`reset` If target reset is detected by the debugger, execute the command.
`timer`	A qualifier used to specify the time interval used with timed events. The minimum interval is 10ms.
`replace`	A qualifier used to specify that this `ONSTATE` command replaces all previous ONSTATE commands for the same event. If this qualifier is not specified, new commands for an event are added to the end of a list of commands to execute when the event happens.
`command`	The debugger command to execute. It can be more than one word.

Description

The `ONSTATE` command executes a given debugger command when a specified event occurs. If no arguments are provided, `ONSTATE` lists out the currently registered commands for each type of event.

Examples

The following examples show how to use `ONSTATE`:

`onstate,tstop,timer:5000 ce 0x8000`

While the debugger has the target stopped at a five-second interval, execute the command `ce 0x8000`.

`onstate,stop,replace`

Delete the event commands associated with the stop event.

`onstate` List the current event commands in the following format:

```
On Start:
  <no commands registered>On Stop:
  <no commands registered>On Start Timed (every 0 msecs):
  <no commands registered>On Stop Timed (every 5000 msecs):
  ce 0x8000On Reset:
  <no commands registered>
```

See also

The following command provides similar or related functionality:

- *BGLOBAL* on page 2-38.

2.3.85 OPTION

The `OPTION` command enables you to change the settings of debugger options for this session, or to display their current settings.

Syntax

`OPTION [option = value]`

where:

`option` Specifies a setting from the list:

`RADIX` The number base used for numeric input and output. The *`value`* must be one of:

`DECIMAL` The default input number base is decimal, base 10, using the digits `0..9`. You can also suffix a decimal number with `t`. This is the default setting.

`HEXADECIMAL`

The default input number base is hexadecimal, base 16, using the digits `0..9` and `a..f`, or `0..9` and `A..F`. You can also prefix a hexadecimal number with `0x` or suffix it with `h`.

Note

It is suggested that you use either the 0x prefix or h suffix for every hexadecimal number. This ensures that the value is valid if you change the radix to decimal. For example, `0x80FF` is always valid, but `80FF` is invalid for a decimal radix.

Also, if you use the `h` suffix, it is suggested that you prefix the hexadecimal number with a zero digit to avoid confusion with symbol names, for example, `0FADEh`.

`OUTDEC` The output number base is decimal, base 10, using the digits `0..9`. This is the default setting.

`OUTHEX` The output number base is hexadecimal, base 16, is prefixed with `0x` and uses the digits `0..9` and `A..F`.

The number base for a particular session can also be set in the workspace options.

`FRAMESTOP`

A flag that controls the behavior of the call stack algorithm. The *`value`* must be one of:

`ON`	The call stack stops when a stack frame is encountered that does not have associated debug information.
`OFF`	The call stack stops when the end of stack is reached or when the stack frame no longer makes sense.

`DEMANDLOAD`

A flag that controls when the debugger symbol table is loaded. The *`value`* must be one of:

`ON`	The debug sections of the executable file are loaded into the debugger symbol table as required, speeding up the target load time. This is the default setting.
`OFF`	The whole symbol table is loaded from the file when the `LOAD` or `RELOAD` commands are issued.

`ENDIANITY`

A flag that indicates the endianness of the target. The *`value`* must be one of:

`LITTLE`	The least significant byte of data is in the lowest address in memory, or appears first in a word in a data stream.
`BIG`	The most significant byte of data is in the lowest address in memory, or appears last in a word in a data stream.

Note

The option changes how the debugger sends and receives data from the target. It cannot be used to change the endianness of the target itself. You must set this value, or the equivalent board file setting, to reflect the target, or the debugger might not work with your target.

`value` Defines the value that you want to assign to the specified option.

Description

The `OPTION` command enables you to change the settings of debugger options for this session, or to display their current settings. If you supply no parameters, the command displays the current settings of various options.

Examples

`option`

Displays the current option settings, for example:

```
RADIX = DECIMAL, OUTHEX
FRAMESTOP = OFF
DEMANDLOAD = ON
ENDIANITY = LITTLE
```

`option radix=hex`

The numerical input base is hexadecimal. The following are valid numbers when the default number base is hexadecimal:

- `0xAB` (AB hex, 171 decimal)
- `0AB` (AB hex, 171 decimal)
- `45` (45 hex, 69 decimal)
- `45t` (45 decimal)
- `45H` (45 hex, 69 decimal).

and the following are not valid:

- `AB` (does not start with a digit)
- `0t45` (`t` must be at the end).

The following example opens a user-defined window with the name `User80` followed by a window named `User50`:

```
>option radix=hex
>vopen 50
>option radix=dec
>vopen 50
```

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *LOAD* on page 2-189
- *PRINTVALUE* on page 2-229
- *SETTINGS* on page 2-263.

2.3.86 OSCTRL

The `OSCTRL` command controls OS Awareness.

Syntax

```
OSCTRL ,qualifier [=value]
```

where:

qualifier Specifies the action, and can be one of:

`enable_rsd`
: Enable RSD.

`disable_rsd`
: Disable RSD.

`properties_rsd`
: Report the current RSD properties on the screen, for example the status of the RSD module, settings as specified in your board file (or `.bcd` file where available), and RSD breakpoints. If you are using the GUI, then the properties are displayed in the Output pane.

value Specifies a file where filters are saved.

Not available in this release.

Examples

The following example shows how to use `OSCTRL`:

```
osctrl,enable_rsd
```

See also

The following commands provide similar or related functionality:

- *AOS_resource-list* on page 2-32
- *BREAKINSTRUCTION* on page 2-61
- *DOS_resource-list* on page 2-129
- *HALT* on page 2-177
- *STOP* on page 2-283.

2.3.87 PATHTRANSLATE

The PATHTRANSLATE command manipulates the host/target name translation list.

Syntax

PATHTRANSLATE [,*qualifier*] [=*path-translation*, ...]

where:

qualifier If specified, must be one of the following:

replace	The current path translation list is to be replaced by the list specified in this command.
default	The current path translation list is to be deleted.
delete	The current path translation list is to be deleted.

If you do not supply a qualifier, the path translation list specified in this command is appended to the current path translation list.

path_translation

A list of path translations, that must be enclosed in quotes ("). The format is:

```
"hpath=tpath"
```

where hpath is the path on the host and tpath is the path on the target. Both hpath and tpath can be a comma-separated list.

Target can be '-' which means a grab of a process with no path prepends the host path.

Description

The PATHTRANSLATE command extends, replaces, or deletes the path translation for a board. Path translation enables the paths on a host computer to be different from those on a remote target.

If you do not provide a path translation in the command, you are prompted to enter a path translation for the current board:

- If you enter this command in the **Cmd** tab of the RealView Debugger GUI, a prompt dialog is displayed and shows any existing translation.
- If you enter this command in the headless debugger, you can press Enter to show the current translation.

If translating from host name to target name, the first target name in the list is used. If translating from target name to host name, the first host name in the list is used.

Examples

The following examples show how to use `PATHTRANSLATE`:

```
pathtranslate "hostpath1=targetpath1","hostpath2=targetpath2"
```

```
pathtranslate "hostpath1,hostpath2,hostpath3=targetpath1,targetpath2"
```

See also

The following commands provide similar or related functionality:

- *LOAD* on page 2-189
- *NAMETRANSLATE* on page 2-206.

2.3.88 PAUSE

The `PAUSE` command waits for a specified number of seconds.

Syntax

`PAUSE [n]`

where:

n	Specifies a period of time, in seconds.

Description

The `PAUSE` command pauses command file reading. It stops execution of commands from the include file for a specified time, or until you indicate that execution can continue.

If you do not supply a parameter, or supply a value of zero, the command waits indefinitely. Execution continues when you press `Return`, `Enter`, or `Cancel`.

If you supply a positive integer, a countdown of seconds from that number to zero is displayed. Execution continues when zero is reached, or earlier if you press `Return`, `Enter`, or `Cancel`.

Note

This command requires that RealView Debugger is connected to a debug target.

Examples

The following examples show how to use `PAUSE`:

`pause 5`	Wait for 5 seconds, or for you to press `Return`, `Enter`, or `Cancel`, and then continue.
`pause`	Wait for you to press `Return`, `Enter`, or `Cancel`.

See also

The following command provides similar or related functionality:
- *WAIT* on page 2-346.

2.3.89 PRINTDSM

The `PRINTDSM` command prints disassembled target memory at a specified address or between a range of addresses.

Syntax

```
PRINTDSM { address | addressrange }
```

where:

address	The address containing the line of code to be disassembled. For example, `0x8000`.
addressrange	The start and end addresses containing the code to be disassembled. For example, `0x8000..0x8FFF` specifies addresses in the address range `0x8000` to `0x8FFF`. See *Specifying address ranges* on page 2-2 for more details.

Description

The `PRINTDSM` command prints disassembled target memory at the specified address, or in the specified address range:

- if you are using the GUI, then output is sent to the **Cmd** tab of the output pane
- if you have a journal file open, the disassembly is also sent to that file.

The output is in the same format as the disassmbly in the **Dsm** tab of the File Editor pane, but without the destination labels.

Note

This command requires that RealView Debugger is connected to a debug target.

Examples

The following examples use the `dhrystone` image to show how to use `PRINTDSM`:

`printdsm 0x8000..0x800F`

Prints a disassembled version of the code from `0x8000` to `0x800F`, for example:

```
00008000 EA000000  B        __scatterload_rt2          <0x8008>
00008004 EA000624  B        __rt_entry                 <0x989c>
00008008 E28FC028  ADR      r12,{pc}+0x30 ; #0x8038
0000800C E89C0C00  LDMIA    r12,{r10,r11}
```

`printdsm main..+16`

Prints a disassembled version of the 16 bytes of code starting from the address of function `main`, for example:

```
000083D0 E92D4FF0  STMFD    r13!,{r4-r11,r14}
000083D4 E24DD06C  SUB      r13,r13,#0x6c
000083D8 E3A00030  MOV      r0,#0x30
000083DC EB00033C  BL       malloc                   <0x90d4>
```

See also

The following commands provide similar or related functionality:

- *DISASSEMBLE* on page 2-122
- *DUMPMAP* on page 2-143
- *DUMP* on page 2-141.

2.3.90 PRINTF

The `PRINTF` command prints formatted text on the screen.

Syntax

`PRINTF "`*`format_string`*`" [,`*`argument`*`]...`

where:

`format_string`

Is a format specification conforming to C/C++ rules with extensions. It might be a text message, or it can describe how one or more arguments are to be presented. See *Format string syntax* for details.

Note

Only the first 256 characters of the string are displayed, even after formatting is applied.

`argument`	Is a list of values that you want displayed in the way described by the specified format.

Description

The `PRINTF` command uses a special format string to write text and numbers to the screen. If you are using the GUI, then they are displayed in the Output pane. It works in a similar way to the ANSI C standard library function `printf()`, with a number of extensions to better support the debugger environment.

Format string syntax

The message in *`format_string`* is a string. If there are no `%` characters in the string, the message is written out and any arguments are ignored. The `%` symbol is used to indicate the start of an argument conversion specification. The syntax of the specification is:

`%[`*`flag`*`][`*`fieldwidth`*`][`*`precision`*`][`*`lenmod`*`]`*`convspec`*

where:

`flag`	An optional conversion modification flag `-`. If specified, the result is left-justified within the field width. If not specified, the result is right-justified.
`fieldwidth`	An optional minimum field width specified in decimal.

precision	An optional precision specified in decimal, with a preceding `.` (period character) to identify it.
lenmod	An optional argument length specifier:

h	a 16-bit value
l	a 32-bit value
ll	a 64-bit value

convspec The possible conversion specifier characters are:

% A literal `%` character.

m The mnemonic for the processor instruction in memory pointed to by the argument. The expansion includes a newline character. The information that is printed includes:

- the memory address in hexadecimal
- the memory contents in hexadecimal
- the instruction mnemonic and arguments
- an ASCII representation of the memory contents, if printable.

H A line from the current source file, where the argument is the line number.

h A line from the current source file, where the argument is a target memory address.

d, i, or u An integer argument printed in decimal. `d` and `i` are equivalent, and indicate a signed integer. `u` is used for unsigned integers.

x or X An integer argument printed in unsigned hexadecimal. `x` indicates that the letters `a` to `f` are used for the extra digits, and `X` indicates that the letters `A` to `F` are used.

c A single character argument.

s A string argument. The string itself can be stored on the host or on the target.

p A pointer argument. The value of the pointer is printed in hexadecimal.

e, E, f, g, or G

A floating point argument, printed in scientific notation, fixed point notation, or the shorter of the two. The capital letter forms use a capital `E` in scientific notation rather than an `e`.

Rules

The following rules apply to the use of the `FPRINTF` command:

- if there are too many arguments, some of them are not printed
- if there are too few arguments (that is, there are more conversion specifiers in the format string than there are arguments after the format string), the string `<invalid value>` is output instead
- if the argument type does not correspond to its conversion field specification, arguments are converted incorrectly.

Examples

The following examples show how to use `PRINTF`:

`printf "Found %d errors\n", ecount`

Print out a message, substituting the value of `ecount`. So, if `ecount` had the value `5`, the message is:

```
Found 5 errors
```

`printf "Completion %%\n", runs`

Print out a message that includes a single percent symbol. The argument `runs` is ignored, so the message is:

```
Completion %
```

`printf "%h\n", #82`

Print out a source file line 82. For example:

```
REG   char            Ch_Index;
```

`printf "Var is %hd.\n", short_var`

Print out the variable `short short_var`. For example:

```
Var is 22.
```

`printf "Instruction1 %m\nInstruction2 %m", 0x100, 0x104`

Print out the disassembly of the contents of location `0x100`, two newlines and the contents of location `0x104`. For example, it might print:

```
Instruction1 000000100 20011410  ANDCS    r1,r1,r0,LSL r4

Instruction2 000000104 20011412  ANDCS    r1,r1,r2,LSL r4
```

```
printf "Average execution time %f secs\n", totaltime / (double)20
```

Print out a message, substituting the value of the expression. So, if `totaltime` had the value `523.3`, the message is:

```
Average execution time 26.165 secs
```

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *FPRINTF* on page 2-169
- *PRINTTYPE* on page 2-227
- *PRINTVALUE* on page 2-229.

2.3.91 PRINTSYMBOLS

The `PRINTSYMBOLS` command displays information about the specified symbol including its name, data type, storage class, and memory location.

Syntax

```
PRINTSYMBOLS [{/C|/D|/E|/F|/M|/R|/T|/W}] [name[*]] [{\|\\|*}]
```

where:

`/C`	Displays functions and labels.
`/D`	Displays data and macros.
`/E`	Displays any symbol declaration conflicts. Mismatch errors occur when global variables are declared with different types in different modules or global functions are declared with different return types or argument counts in different modules.
`/F`	Displays symbols in all roots (all contexts). All matching names in all roots are shown.
`/M`	Displays modules and module names.
`/R`	Displays reserved symbols, registers, and internal variables.
`/T`	Displays types.
`/W`	Displays symbols in wide format (names only).
`name`	Specifies the symbolic unit. The wildcard character (`*`) can be used to match the first zero or more letters of a name. The `*` must be the last character in the partial name.
`*`	An asterisk as the only parameter displays all symbols in the current context.
`\`	Displays information about all modules.
`\\`	Displays information about debugger symbols.

Description

The PRINTSYMBOLS command displays information about the specified symbol including its name, data type, storage class, and memory location. If you want to see all modules in your current root, use only \ and \\. If you want to see all symbols in a particular function or module, append \ to the module name.

PRINTSYMBOLS with no options specified acts the same as the CONTEXT command. Also, PRINTSYMBOLS /F acts the same as CONTEXT /F.

Note

The symbol name must be specified in the correct case, even when a wildcard is used.

Examples

The following examples show how to use PRINTSYMBOLS:

`printsymbols funct1\`

Prints the names of all symbols within funct1, for example, all local variables.

`printsymbols /m *`

Prints the names of all modules in the image. For example, for the dhrystone image this command prints:

```
@dhrystone\\DHRY_H    : Codeless Include File.
@dhrystone\\TIME_H    : Codeless Include File.
@dhrystone\\DHRY_1    : High level module.
                        Code section = 0x00008278 to 0x00008FB3
                        Code section = 0x0000D8BC to 0x0000D8BF
@dhrystone\\DHRY_2    : NON-LOADED module.
                        Code section = 0x0000807C to 0x0000826F
@dhrystone\\STARTUP_S : Assembly level module.
                        Code section = 0x00008000 to 0x00008007
@dhrystone\\SCATTER_S : Assembly level module.
                        Code section = 0x00008008 to 0x0000807B
@dhrystone\\SYSAPP    : Assembly level module.
                        Code section = 0x00008FB4 to 0x00008FF3
                        Code section = 0x0000AE20 to 0x0000AEE7
                        Code section = 0x0000D07C to 0x0000D093
@dhrystone\\HEAPALLOC : Assembly level module.
                        Code section = 0x00008FF4 to 0x000090EF
                        Code section = 0x000098E4 to 0x000098EB
                        Code section = 0x0000AF14 to 0x0000AF33
...
```

Alias

`PS` is an alias of `PRINTSYMBOLS`.

See also

The following command provides similar or related functionality:

- *CONTEXT* on page 2-102.
- *PRINTTYPE* on page 2-227.

2.3.92 PRINTTYPE

The `PRINTTYPE` command displays language type information for a symbol.

Syntax

`PRINTTYPE {symbol_name | expression}`

where:

symbol_name	Specifies the name of a symbol.
expression	Specifies a debugger expression.

Description

The `PRINTTYPE` command displays language type information for a symbol or debugger expression. The information is displayed in a style similar to the source language.

Note

The symbol name must be specified in the correct case, even if a wildcard is used for part of the name.

Examples

The following examples show how to use `PRINTTYPE`:

`printtype Enumeration`

Shows details of the **`enum`** type `Enumeration`, defined by the `dhrystone` image:

```
typedef enum Enumeration
 {
 , Ident_1:0  Ident_2:1, Ident_3:2, Ident_4:3, Ident_5:4
 } Enumeration;
 -- Defined within module DHRY_H
```

`printtype ptr->databuf`

Shows type details of a field referenced by the pointer `databuf`.

Alias

`PT` is an alias of `PRINTTYPE`.

See also

The following commands provide similar or related functionality:

- *ADD* on page 2-18
- *BROWSE* on page 2-87
- *DELETE* on page 2-114
- *PRINTF* on page 2-220
- *PRINTSYMBOLS* on page 2-224.

2.3.93 PRINTVALUE

The `PRINTVALUE` command prints the value of a variable or expression.

Syntax

```
PRINTVALUE [{/H|/MB|/R|/S|/T}] {expression | expression_range}
```

where:

`/T`	Displays the value in decimal format.
`/H`	Displays the value in hexadecimal format.
`/MB`	Displays multibyte characters using the current encoding, for example UTF-8. You must use the GUI to set up the character encoding.
`/R`	Suppresses the display of the address when you specify a variable in an image.
`/S`	Suppresses the display of characters in the string, but displays the character pointer.
`expression`	Specifies an expression to be displayed. If you are using the GUI, then the expression is displayed in the Output pane.
`expression_range`	Specifies an expression range to be displayed. If you are using the GUI, the expressions range is displayed in the Output pane.

Description

The `PRINTVALUE` command prints to the screen the value of a variable or expression using its natural type for formatting. It can display all of aggregate types, such as structures, and expressions can be type cast to display it in a different format. All values that make up a complex type are printed. If you are using the GUI, then they are displayed in the Output pane.

Each value within an `expression_range` is displayed according to the base type if one exists. All expressions printed with this command are displayed according to their type. If the type of the expression is unknown, it defaults to type byte.

The `PRINTVALUE` command runs synchronously unless access to target memory is required and background access is not possible. Use the `WAIT` command to force it to run synchronously.

The following messages can be displayed by the PRINTVALUE command:

`<ENUM: xx>`	Invalid enum value, xx = value.
`<INFINITY>`	Floating-point value is infinity.
`<NAN>`	Not a number. A floating-point error.

Examples

The following examples show how to use PRINTVALUE:

`printvalue /mb pchUTF8`

Prints the multibyte character variable pchUTF8, encoded in UTF-8, as multibyte characters. Without the /mb switch the characters are displayed as escaped characters.

`printvalue *Ptr_Glob`

The command can be used to print the full contents of a record, for example this instance from a run of dhrystone:

```
printv *Ptr_Glob
0x00011540 = {Ptr_Comp=(record *)0x00011508,Discr=Ident_1,variant={var_1=
          {Enum_Comp=Ident_3,Int_Comp=17,Str_Comp="DHRYSTONE PROG
           RAM, SOME STRING"},var_2={E_Comp_2=Ident_3,Str_2_Comp="C
           \x02\xC7\x11"},var_3={Ch_1_Comp='\x02',Ch_2_Comp='C'}}}
```

Note

For the same expression, CEXPRESSION prints the address, not the full value:

```
> ce *Ptr_Glob
  Result is: data address 0x00011540
```

`p Ptr_Glob` Printing the value of the pointer tells you the address of the pointer, its type and the value stored there:

```
0x0000EBBC = (record *)0x00011540
```

Note

For the same expression, CEXPRESSION prints the value of the pointer, but not its type and address:

```
> ce Ptr_Glob
  Result is: data address 0x00011540
```

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *MONITOR* on page 2-204.

2.3.94 PROPERTIES

`PROPERTIES` is an alias of `SETTINGS` (see page 2-263).

2.3.95 PS

`PS` is an alias of `PRINTSYMBOLS` (see page 2-224).

2.3.96 PT

`PT` is an alias of `PRINTTYPE` (see page 2-227).

2.3.97 QUIT

The `QUIT` command causes the debugger to exit.

Syntax

```
QUIT [Y]
```

where:

`Y`	Exits the debugger without displaying a confirmation dialog. Include this when using the `QUIT` command in a batch file.

Description

The `QUIT` command exits the debugger. It displays a dialog box where you can confirm the operation.

If you have any unsaved changes, you are prompted to save these before the debugger exits.

Examples

The following examples show how to use `QUIT`:

`quit`	Exits the debugger. Displays a dialog box where you can choose to confirm or abort the operation. If you choose to exit, the debugger warns of any unsaved changes.
`quit y`	Exits the debugger without additional confirmation. However, the debugger warns of any unsaved changes.

2.3.98 READBOARDFILE

The READBOARDFILE command reads the specified board file.

Syntax

READBOARDFILE [,auto] [=*board-filename*]

where:

auto	Is an optional qualifier. If you specify auto the command does not read the specified board file if it is the same as the last one read.
board-filename	Specifies the name of the board file to read. It must be enclosed in double quotes. You can include one of more environment variables in the filename. For example, if MYPATH defines the location C:\Myfiles, you can specify: readboardfile ="$MYPATH\\gizmo.brd"

Description

The READBOARDFILE command reads the specified board file. If you do not specify a board file, the command rereads the current board file. If you do not specify a board file and no board file has been read, the command reads the default rvdebug.brd.

The READBOARDFILE command runs synchronously.

Examples

The following example shows how to use READBOARDFILE:

```
readboardfile ="c:\sources\gizmo.brd"
```

Read the file gizmo.brd into memory, replacing the current file.

See also

The following commands provide similar or related functionality:

- *ADDBOARD* on page 2-21
- *DELBOARD* on page 2-113
- *EDITBOARDFILE* on page 2-146.

2.3.99 READFILE

The `READFILE` command reads a file into target memory.

Syntax

`READFILE ,{OBJ|raw|rawb|rawhw|ascii[,`*`opts`*`]}` *`filename`* `[=`*`address`*`]`

where:

`OBJ`	The file is an executable file in the standard target format. RVDK can read files in either the ELF format or an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The proprietary file format for each version of the RVDK is restricted to the permitted device. This is referred to as the ATPE_*Custom*. There are no *`opts`* supported for this file type.
`raw`	Write the file as raw data, one word per word of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`rawb`	Write the file as raw data, one byte per byte of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`rawhw`	Write the file as raw data, one halfword per halfword of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`ascii`	The file is a stream of ASCII digits separated by whitespace. The interpretation of the digits is specified by other qualifiers (see the *`opts`* qualifier). The starting address of the file must be specified in a one line header in one of the following ways:

`[`*`start`*`]`	The start address.
`[`*`start`*`,`*`end`*`]`	The start address, a comma, and the end address.
`[`*`start`*`,+`*`len`*`]`	The start address, a comma, and the length.
`[`*`start`*`,`*`end`*`,`*`size`*`]`	The start address, a comma, the end address, a comma, and a character indicating the size of each value, where `b` is 8 bits, `h` is 16 bits and `l` is 32 bits. This is the format used by the `WRITEFILE` command.

If the size of the items in the file is not specified, the debugger determines the size by examining the number of white-space separated significant digits in the first data value. For example, if the first data value was `0x00A0`, the size is set to 16-bits.

`opts` Optional qualifiers available for use with the `ascii` qualifier:

`byte`	The file is a stream of 8-bit values that are written to target memory without extra interpretation.
`half`	The file is a stream of 16-bit values.
`long`	The file is a stream of 32-bit values.
`gui`	You are prompted to enter the file type with a dialog.

——— **Note** ———

This option has no effect in the headless debugger.

`filename` The name of the file to be read. This can be a file that you have written with the `WRITEFILE` command.

You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myfiles`, you can specify:

```
readfile,raw "$MYPATH\\myfile.dat" 0x8000..0x8100
```

`address` The starting address of target memory must be given if not in the file. If the file contains a start address, the parameter becomes a signed offset that is added to the file starting address. This is useful for loading Flash images.

Description

The `READFILE` command reads a file, performs a format conversion if necessary on its contents, and loads the resulting information into target memory.

The types of file and file formats supported depend on the target processor and any loaded DLLs. The type of memory assumed depends on the target processor. For example, ARM processors have byte addressable memory.

Examples

The following examples show how to use `READFILE`:

```
readfile ,OBJ "c:\temp\file.exe"
```

Reads the contents of the named executable file into memory at its specified start address.

```
readfile ,ascii,long "c:\temp\file.txt" =0x2000
```

Reads the contents of the named text file into memory, writing values as words using the target endianness to translate values in the file into bytes in target memory. The file is written starting at address `0xA000`, because the file contains a start address and an offset is specified. The file contents can look, for example, like this:

```
[0x8000,0x9000,l]
E28F8090 E898000F E0800008 E0811008
E0822008 E0833008 E240B001 E242C001
E1500001 0A00000E E8B00070 E1540005
...
```

See also

The following commands provide similar or related functionality:

- *FILL* on page 2-161
- *FLASH* on page 2-164
- *LOAD* on page 2-189
- *SETMEM* on page 2-257
- *VERIFYFILE* on page 2-338
- *WRITEFILE* on page 2-351.

2.3.100 REEXEC

`REEXEC` is an alias of `RESTART` (see page 2-245).

2.3.101 RELOAD

The `RELOAD` command loads a linked program image containing program code and data.

Syntax

```
RELOAD [{,qualifier...}] [{filename | file_num}] [=task]
```

where:

qualifier If specified, *qualifier* must be one of the following:

`all`	Loads all the files in the file list.
`symbols_only`	Reloads the symbols only, not the executable image.
`image_only`	Reloads the executable image only, not the symbols.
`force`	Forces the load to proceed even if it might be aborted because, for example, the file being loaded overlaps a file already loaded.

filename | *file_num*

Specifies a file to be reloaded. If you do not specify a file, the whole process is reloaded.

Use the `DTFILE` command to list details of the file or files that are associated with the current connection (see *DTFILE* on page 2-137). The details include:

- the file number, which is shown at the start of the output for each file listed by the text `File` *file_num*
- the filename and path.

`task` Specifies the task that is to start. This parameter is required only when the target is running multiple tasks.

Description

The `RELOAD` command loads or reloads an absolute file image containing program code and data. You can load a specified file, or one or more files from the file list. The PC is reset to the start location.

If any file being reloaded is already loaded, it is unloaded before being loaded again. If the symbols for a given file are already loaded, they are not reloaded unless the file modification date has changed.

You can reload symbols only, or the image only. For details see the descriptions of the command qualifiers.

The effect of reloading the system file is defined by the vehicle.

——— **Note** ———

If you reload an image that requires arguments, you must use the `ARGUMENTS` command to specify them before running the image. Alternatively, use the `LOAD` command and specify the arguments as part of that command.

———————————

See also

The following commands provide similar or related functionality:

- *ADDFILE* on page 2-23
- *ARGUMENTS* on page 2-34
- *DTFILE* on page 2-137
- *LOAD* on page 2-189
- *READFILE* on page 2-235
- *UNLOAD* on page 2-333.

2.3.102 RESET

The RESET command performs or simulates a target processor reset.

Syntax

RESET [,cleanup] [=*resource*]

where:

cleanup	Use this command qualifier only with operating systems that support it. Its purpose is to cleanup thread states and other OS issues.
resource	Specifies the target processor that is to be reset, for example, @ARM7TDMI_0@RealView-ICE. You must specify @RealView-ICE only if the target identifier is not unique. For example, you might have another @ARM966E-S_0 target on an RVI-ME connection.

Description

This command is used to reset the target processor and peripherals on the board. If an actual hardware reset is not possible, the command places the processor in a state that is as close as possible to the hardware reset state. The actual behavior varies from one processor type to another and from one vehicle type to another. Check with the manufacturer for details. Variables are not reset to their original values, because memory is not re-initialized

The RESET command runs synchronously.

Alias

WARMSTART is an alias of RESET.

Examples

The following examples show how to use RESET:

```
reset @ARM966E-S_0@RealView-ICE
```

Resets the ARM966E-S processor on the RealView-ICE connection.

```
reset,cleanup @ARM966E-S_0
```

Resets the ARM966E-S processor, and cleans up the thread states and any other OS issues on the processor.

See also

The following command provides similar or related functionality:

- *RESTART* on page 2-245.

2.3.103 RESETBREAKS

The RESETBREAKS command resets breakpoint pass counters and *and-then* conditions.

Syntax

```
resetbreaks [,h] [break_num,...] [=thread,...]
```

where:

h	Do not use this qualifier. It is for debugger internal use only.
break_num	Specifies one or more breakpoints to have their pass counters reset to zero. You identify breakpoints by their position in the list displayed by the DTBREAK command (see page 2-135).
thread	Specifies one or more threads to which this command applies. Other threads remain unaffected. If you do not supply this parameter, then breakpoints on all threads are reset. You do not have to supply this parameter if the processor has only a single execution thread or the RTOS extension is not enabled.

Description

The RESETBREAKS command resets breakpoints pass counters. The pass counters are the counts of the number of times breakpoints have been triggered, as shown by the DTBREAK command (see page 2-135). It also resets the *and* and *and-then* condition state so that the first breakpoint is again required before the second can trigger. For more information on *and* and *and-then* conditions see *BREAKEXECUTION* on page 2-52.

If you issue a RESETBREAKS command without specifying a breakpoint number, the pass counter, *and* and *and-then* conditions for all the current pass counters are reset to zero.

You might typically issue a RESETBREAKS command after a RELOAD command, so that the counts all begin again from zero when you restart execution.

Examples

The following examples show how to use RESETBREAKS:

`resetbreaks 4,6,8`	Resets the pass counters and conditions of the fourth, sixth, and eighth breakpoints in the current list of breakpoints.
`resetbreaks =2`	Resets all the pass counters and conditions in thread 2.

Alias

`RSTBREAKS` is an alias of `RESETBREAKS`.

See also

The following commands provide similar or related functionality:

- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *CLEARBREAK* on page 2-94
- *DTBREAK* on page 2-135
- *RELOAD* on page 2-239.

2.3.104 RESTART

The `RESTART` command resets the PC to the program starting address.

Syntax

`RESTART [=task]`

where:

`task`	Specifies the task that is to start. This parameter is required when the target is running multiple tasks and the RTOS extension is enabled.

——— **Note** ———

This is not avalaible in this release.

———————————

Description

The `RESTART` command resets the PC to the program starting address, so that the next `GO`, `STEP` or `GOSTEP` command restarts execution at the beginning of the program. The `RESTART` command does not reset the values of variables, the stack pointer is not reset and breakpoints are not cleared. If required, `RESTART` can be configured to reload the image using the `SETTINGS` command. All declared I/O ports are unaffected. You can use the `ARGUMENTS` command (see page 2-28) to change the arguments passed to the process for a restart.

——— **Note** ———

- If the program relies on the initial values of variables in initialized data areas, and those variables are modified during program execution, then using `RESTART` to rerun the program fails.
- The `RESTART` command might behave differently if you are using the RTOS extension to RealView Debugger. See the instructions for the specific RTOS extension for more details.

———————————

The `RESTART` command runs synchronously.

Alias

`REEXEC` is an alias of `RESTART`.

See also

The following commands provide similar or related functionality:

- *GO* on page 2-173
- *RELOAD* on page 2-239.

2.3.105 RSTBREAKS

`RSTBREAKS` is an alias of `RESETBREAKS` (see page 2-243).

2.3.106 RTOS action commands

RTOS action commands. See *AOS_resource-list* on page 2-32 for details.

2.3.107 RTOS resource commands

RTOS resource commands. See *DOS_resource-list* on page 2-129 for details.

2.3.108 RUN

The `RUN` command starts execution using a specific mode, or sets the default mode used by the `GO` command.

Syntax

```
RUN [,setdefault][,mode]
```

where:

`setdefault`	Set the default mode for the `GO` command to the mode specified by this command, but do not start execution.
`mode`	If specified, must be one or more of the following:

`debug` **or** `normal`	Run with breakpoints active. This is the default mode.
`clock` **or** `benchmark`	Run programs with breakpoint timing hardware enabled. This option is only available on some targets.
`free` **or** `user`	Run at full speed, with breakpoints disabled. Depending on the target, hardware, this might not be any faster than `normal` mode.

Description

If supported by the target, the `RUN` command starts execution using a specific mode, or sets the default mode used by the `GO` command. If you supply no parameters, `RUN` displays the current mode.

Examples

The following examples show how to use `RUN`:

`run,setdefault,normal`

Set the default run mode to `normal`, so that the next GO command for this connection runs the target in the normal way.

`run,free` Run the target using the `free` run mode.

See also

The following command provides similar or related functionality:

- *GO* on page 2-173

- *HALT* on page 2-177
- *STOP* on page 2-283.

2.3.109 SCOPE

The `SCOPE` command specifies the current module and procedure scope.

Syntax

```
SCOPE /F
```

```
SCOPE root_name\\
```

```
SCOPE [root_name\\] module_name
```

```
SCOPE [[root_name\\] module_name\] {function_name | (expression) | @stack_level
| #line_number}]
```

where:

`/F`	Selects the first module of the next root.
`root_name`	Specifies the name of a root (for example, `@sieve`).
`module_name`	Specifies the name of a module (for example, `SIEVE`).
`function_name`	Specifies the name of a function (for example, `proc1`).
`expression`	Specifies an expression specifying the location of a calling function.
`stack_level`	Specifies a stack level.
`line_number`	Specifies a high-level line number.

Description

The `SCOPE` command specifies the current module and procedure scope. This determines the current context. The current context determines how local variables are accessed and what symbol qualification is required. The following context types are supported:

- the current PC
- a specific module, function, or source file line
- a stack frame position
- auto-set, used when the debugger is in source mode and the PC is not in a source view context, for example when the program is at the entry point.

The `SCOPE` command can change the default root, module, procedure, line number, or stack level, but it does not change the PC.

To return the scope to display source at the current PC location, use `SCOPE` with no parameters. To display the current scope, use the `CONTEXT` command.

The current root and module is the default when line numbers and local symbols are referenced without a module or procedure qualifier. For example, if line number 3 is entered on the command line as `#3`, it is interpreted as `default_module\#3`. The new source file or disassembly is shown in the Code window.

The `SCOPE` command runs asynchronously. Use the `WAIT` command to force it to run synchronously.

Examples

The following examples show how to use `SCOPE`:

`scope #155`	Set the current context to line 155 in the current module (file). `Scoped to: (0x01000560): DHRY_1\main Line 155`
`sc \\DHRY_1`	Set the current context to the start of the file dhry_1.c. `Scoped to: (0x010002BC): DHRY_1\main Line 78`
`sc @1`	Set the scope to the stack frame of the calling function.
`sc`	Return the current context to the execution point. `At the PC: (0x01000544): DHRY_1\main Line 152`

See also

The following commands provide similar or related functionality:

- *CONTEXT* on page 2-102
- *PRINTVALUE* on page 2-229
- *WHERE* on page 2-348.

2.3.110 SEARCH

The SEARCH command searches memory for a specified value or pattern.

Syntax

SEARCH [{/B|/H|/W|/8|/16|/32}] [/R] [*address_range* [={*expression* | *expression_string*}]]

where:

/B, , /8	Sets the display format to byte (8 bits). If the processor naturally addresses bytes (for example, ARM7TDMI) then this is the default setting.
/H, , /16	Sets the display format to halfword (16 bits).
/W, , /32	Sets the display format to word (32 bits).
/R	Continues to search for the specified expression displaying each match until the end of the block or until the STOP button is used.
address_range	Specifies the range of addresses to be searched. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.
expression	Specifies the value to search for.
expression_string	Specifies the pattern to search for.

Description

The SEARCH command searches a memory area for the specified value or pattern string. When it is found, the debugger stops searching and displays the address where the expression was found.

If they do not fit the specified size evenly, all expressions in an expression string are padded or truncated to the size specified by the size qualifiers. If you do not specify an expression or expression string, the debugger searches the memory area for zeros. If you issue a SEARCH command without parameters, the debugger continues searching through the originally specified address range starting from where the last match was found.

The SEARCH command runs synchronously.

Examples

The following examples show how to use `SEARCH`:

`search 0x1000..0x2000 =122`

Search for the first occurrence of the byte value `122` (ASCII `z`), in the 4KB block of memory starting at `0x1000`.

`search /r 0x1000..0x2000 =163`

Display all occurrences of the byte value `163` (ASCII `£`) in the 4KB block of memory starting at `0x1000`.

`search 0x1000..0x2000 ="-help"`

Search for the first occurrence of the string `-help` in the 4KB block of memory starting at `0x1000`.

See also

The following commands provide similar or related functionality:

- *MEMWINDOW* on page 2-201
- *SETMEM* on page 2-257.

2.3.111 SETFLAGS

The `SETFLAGS` command is reserved for internal use by the RealView Debugger.

2.3.112 SETMEM

The `SETMEM` command changes the contents of memory to a specified value.

Syntax

<u>SETM</u>EM [{/B|/H|/W|/8|/16|/32}] *address* [={*expression* | *expression_string*}]

where:

`/B, , /8`	Sets the access size to byte (8 bits). This is the default setting.
`/H, , /16`	Sets the access size to halfword (16 bits).
`/W, , /32`	Sets the access size to word (32 bits).
`address`	Specifies the memory address where the contents are to be changed.
`expression`	Specifies an expression to be evaluated to a value and placed into the specified memory address.
`expression_string`	Specifies the string pattern to be placed into the specified memory address.

Description

The `SETMEM` command changes the contents of the specified memory location to the value or values defined by `expression` or `expression_string`. `SETMEM` is used to set assembly-level memory. For example, you can use it to work around a section of code that is producing incorrect results by changing variables to the correct values. If you do not specify a value then the interactive setting dialog is displayed.

An expression string is a list of values separated by commas. ASCII characters enclosed in quotation marks are treated as an array of characters, and with the `/H` and `/W` qualifiers are each expanded to 2 or 4 bytes. All `expressions` in an expression string are padded or truncated to the size specified by the size qualifiers (`/B`, `/H`, `/W`).

——— Note ———

The `SETMEM` command does not recognize variable typing, so you must ensure the expression size qualifier is compatible with the variable type.

———————————

The `SETMEM` command runs synchronously unless background access to target memory is supported. Use the `WAIT` command to force it to run synchronously.

Examples

Assuming the following definitions:

```
    int count=2, buf[8];
    int *ptr = buf;
```

And the following memory map:

```
0x10200 : 0x00000002   count
0x10204 : 0x00000000   buf
0x10224 : 0x00010204   ptr
```

The following two statements both set the value of `count` to 5:

```
setmem /32 &count=5
setmem /32 0x10200=5
```

The following two statements both set the value of `buf[0]` to `0x40`:

```
sm /W 0x10204 =0x40
sm /W ptr     =0x40
```

——— **Note** ———

The command `SM count=5` sets the memory location addressed by the value of `count` to 5, leaving the contents of `count` unchanged.

Alias

`SM` is an alias of `SETMEM`.

See also

The following commands provide similar or related functionality:

- *CEXPRESSION* on page 2-91
- *FILL* on page 2-161.

2.3.113 SETREG

The SETREG command changes the contents of a register, status flag, or a special target variable such as the cycle count.

Syntax

```
SETREG [@register_name [=value]]
```

where:

register_name

Specifies a register. Register names begin with an at sign (@).

value Defines the value to be placed in the register.

Description

This command changes the contents of a register, status flag, or a special target variable such as the cycle count. For more details, see *Referencing reserved symbols* on page 1-15.

Note

If you use this command in the headless debugger, you must supply a *register_name*. Using this command in the headless debugger without any parameters has no effect.

Register names

You can set the value of any register, or register bit-field, that is defined by an active .bcd file. To link a relevant definition file to the current connection, use Connection Properties window to set the BoardChip name for the connection.

You can view the currently defined registers using the PRINTSYMBOLS command (see *PRINTSYMBOLS* on page 2-224), for example:

```
PRINTSYMBOLS /r *
```

This also displays any reserved symbols and internal variables that are currently defined.

By defining new registers in a .bcd file, you can extend the register list to, for example, include peripheral control registers for your target.

Note

Some processors and peripherals have some read-only registers. These cannot be written to with `SETREG`.

Command line usage

You can set the value of registers defined in a board chip file or by the processor model, by prefixing the register name with the @ symbol and assigning it a value. The value can include program and debugger symbols and debugger expressions.

Note

Change the values of processor and device registers with care. Compilers and operating systems do not always use registers in the expected manner.

Fully Interactive register setting

In the GUI, if you supply no parameters, the `SETREG` command displays the Interactive Register Setting dialog where you can specify a register and a value, shown in Figure 2-3. The Register drop-down list contains the names of recently used registers. To select other register names, click either **Next Reg** or **Prev Reg**. The current value of the register is displayed in the Value field, in both unsigned hexadecimal and in signed decimal.

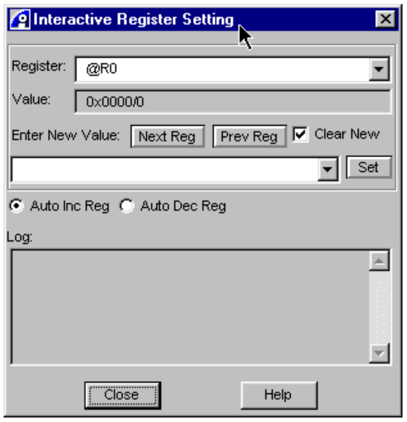


Figure 2-3 Interactive Register Setting dialog

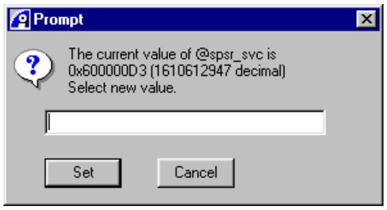
Enter a new value in the combo-box beneath **Enter New Value** and then click **Set**. The **Log** tab displays the changes you have made.

Check **Clear New** to clear the **Enter New Value** field after setting a register with **Set**. If **Clear New** is unchecked, the value you enter remains in the field and you can set multiple registers with repeated clicks on **Set**.

Click **Auto Inc Reg** or **Auto Dec Reg** to select whether, after clicking **Set**, the next higher or next lower numbered register is selected.

Partly Interactive register setting

If you supply only a register name, the `SETREG` command displays a prompt, shown in Figure 2-4, enabling you to enter a new value for that register.



The current value of @spsr_svc is 0x600000D3 (1610612947 decimal) Select new value.

Figure 2-4 Register value prompt

Enter the value in the text field and click **Set** to change the register, or click **Cancel** to abort the command.

Alias

`SR` is an alias of `SETREG`.

Examples

The following examples show how to use `SETREG`:

`setreg @r3=0x50`

Write the value `0x50` to processor register `R3`.

`setreg @spsr_svc`

Display a prompt, shown in Figure 2-4, containing the current value of ARM processor register `SPSR_SVC` (saved program status register, supervisor mode). Use the text box to enter a new value.

`setreg @v=1`

Set the ALU overflow flag in the current program status register.

`setreg` Invoke the Interactive Register Setting dialog shown in Figure 2-3 on page 2-260.

See also

The following commands provide similar or related functionality:

- *ADD* on page 2-18
- *CEXPRESSION* on page 2-91
- *PRINTSYMBOLS* on page 2-224.

2.3.114 SETTINGS

The SETTINGS command enables you to define target settings.

Syntax

```
SETTINGS [{default | option_list}]
```

where:

`default`	Causes all settings to revert to their default values.
`option_list`	A list of option names and values. Each option-value pair consists of a setting name, an equals sign, and a value. The available option names and values are described in *List of options* on page 2-264. You can specify multiple options in the list by separating each option-value pair with a colon.

Description

The SETTINGS command enables you to define settings (properties) for target support. These options have equivalent settings in the Project Properties and Workspace of the GUI. See the individual option description in *List of options* on page 2-264 for the equivalent GUI setting.

Note

If you want to change other Project Properties and Workspace settings, you must use the GUI. See the *RealView Developer Kit v2.2 Debugger User Guide* for more details.

If the only parameter is the `default` qualifier, then all the settings revert to their default values. If you supply no parameters, the command displays the current values of settings for which a default value is defined.

Each setting is defined in the form of *`name`*`=`*`value`*, and multiple settings can be changed using a colon (`:`) as a separator.

List of options

The standard option names are:

`loadact={default|noimask|reset|pre_reset}`

Action on load:

`default`	Normal load image behavior. For ARM processors, the processor is placed in ARM state and supervisor mode with interrupts disabled.
`noimask`	Do not change the processor status register. For example, on ARM processors, the default modification of CPSR is not performed.
`reset`	Reset the processor after the load, to perform a start from reset.
`pre_reset`	Reset the processor before the load.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Load_act`.

`pcset={auto|never|always|user_default}`

When to change the PC when loading an image:

`auto`	Normal, default behavior. The PC is modified if there is an entry point specified in the image file and: • an image file without symbols is loaded and the `LOAD` option `/SP` is used • an image file with symbols is loaded.
`never`	Do not set PC.
`always`	Always set the PC. If no entry point is specified in the image, the PC is set to the reset vector (normally to address `0`).
`user_default`	Always set the PC to the value specified in the `pcdefault` setting.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Set_pc`.

`pcdefault=`*`value`*

User default PC value. This value is written to the target PC when an image file is `LOAD`ed or `RELOAD`ed and the `pcset` option is `user_default`.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Default_pc`.

`regset={@regname=value;...}`

Set the values of registers before the image is loaded. Use the format `@regname=value` for each register, and separate multiple assignments with a semicolon.

See *Reserved symbols* on page 1-14 for more information on possible register names. Also, see `SETREG` on page 2-259.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Load_reg_set`.

`restart={set_pc|reload|reload_data}`

Defines the action of the `RESTART` command. The possible values are:

`set_pc` Set the PC to the entry point of the image.

`reload` Reload the image as for `RELOAD`. The options relating to `RELOAD`, `loadact` and `pcset`, also apply.

`reload_data`

Reload only the initialized data of the image. The options relating to `RELOAD`, for example `loadact` and `pcset`, also apply.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Restart_act`.

`restart_reset={True|False}`

Reset on restart:

`True` Reset the processor on `RESTART`, in addition to any other actions.

`False` Do not reset the processor on `RESTART`.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Restart_reset`.

`verify={none|fast|full}`

Verify the image `LOAD`ed or `RELOAD`ed. The possible values are:

`none` Do not verify the image.

`fast` Perform the normal image verify. The first and last words of every image file section written to the target are read and checked. For individual sections larger than about 2KB, then the first and last words of each 2KB block of the section is checked. This is the default.

`full` Read and check every byte of the image.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Verify`.

`verify_warns={True|False}`

Determines what happens if an image file verification failure occurs:

`True`	The failure is treated as a warning and operation continues.
`False`	The failure is an error and the image load is aborted. This is the default.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Verify_warns`.

`fillstack=`*`value`*

Define a value to fill stack memory with before the program starts. This is not used for ARM processor image files.

Equivalent Project Properties setting: None for ARM processors.

`fillheap=`*`value`*

Define a value to fill memory defined as heap. This is not used for ARM processor image files.

Equivalent Project Properties setting: None for ARM processors.

`fillundefined=`*`value`*

Define a value to fill unused words of memory, such as words between each section of the image in memory. This is not used for ARM processor image files.

Equivalent Project Properties setting: None for ARM processors.

`imagecache_enabled={True|False}`

The image cache stores debug information from the image rather than from physical memory. This enables RealView Debugger to access the debug information if it cannot access the physical memory, for example, when the target is running. This is especially important when tracing, so that you do not have to stop the target for RealView Debugger to collect or decompress trace information.

However, you might want to disable image caching if, for example, you have self-modifying code, which can lead to incorrect information being displayed.

Also, if the image cache is disabled, you cannot view the disassembly of your image when the target is running.

Enables and disables the image cache on a per-project basis:

`True`	Enables the image cache.
`False`	Disables the image cache. This is the default.

Equivalent Project Properties setting: `SETTINGS`, `Image_load`, `Imagecache_enable`.

——— Note ———

Image caching is enabled by default.

———————————

`disasm={default|standard|alternate}`

Set the disassembly mode. The possible values are:

`default` Attempt to auto-detect the disassembly mode.

For ARM architecture processors, select from ARM, Thumb or bytecodes, using information from the image file where available.

`standard` Select the standard, normal instruction disassembly mode.

For ARM architecture processors, select ARM state (32-bit) instructions.

`alternate` Select the standard, normal instruction disassembly mode.

For ARM architecture processors, select Thumb state (16-bit) instructions.

Equivalent Workspace setting: `DEBUGGER`, `Disassembler`, `Format`.

`dsmvalue={True|False}`

Selects whether the instruction code is displayed in disassembly listings. The possible values are:

`True` Disassembly listings include the instruction opcode, along with the instruction memory address and mnemonics.

`False` Disassembly listings do not include the instruction opcode.

Equivalent Workspace setting: `DEBUGGER`, `Disassembler`, `Instr_value`.

Additional options might be implemented for particular target interfaces. See the target interface documentation for more information.

Alias

`PROPERTIES` is an alias of `SETTINGS`.

Examples

The following examples show how to use `SETTINGS`:

```
settings regset=@r2=1;@spsr_svc=0xd0
```

Set the value of processor the ARM processor register R2 and SVC mode SPSR whenever an image is loaded or reloaded.

```
settings pcset=use:pcdefault=0x8000
```

When any image is loaded or reloaded, set the PC to `0x8000`.

```
settings loadact=reset
```

After an image is loaded or reloaded, reset the processor (in hardware). This is useful when the image has been constructed to run from target reset.

```
settings verify=full:verify__warns=true
```

When an image is loaded, check every byte written to the target but do not consider a difference in the value read back to be an error.

See also

The following commands provide similar or related functionality:

- *DISASSEMBLE* on page 2-122
- *LOAD* on page 2-189
- *RELOAD* on page 2-239
- *OPTION* on page 2-211
- *RESET* on page 2-241
- *RESTART* on page 2-245.

2.3.115 SHOW

The `SHOW` command displays the source code of a specified debugger macro.

Syntax

`SHOW macro_name [{,windowid | ,fileid}]`

where:

`macro_name`	Specifies the name of the macro to be displayed.
`,windowid | ,fileid`	Identifies the window or file where the macro is to be directed. This must be a user-defined *`,windowid`* or *`,fileid`*. See *Window and file numbers* on page 1-5 for details. If you do not supply a *`,windowid`* or *`,fileid`* parameter, or there is no window or file associated with the ID, the macro is displayed on the screen. If you are using the GUI, then the macro is displayed in the Output pane.

Description

The `SHOW` command displays the source code of a specified macro. See the macros chapter in the *RealView Developer Kit v2.2 Debugger User Guide* for more information.

Examples

The following example shows how to use `SHOW`:

```
show mac ,50
```

Display the contents of a macro called `mac` in window number 50.

See also

The following commands provide similar or related functionality:

- *DEFINE* on page 2-110
- *INCLUDE* on page 2-181
- *MACRO* on page 2-195.

2.3.116 SINSTR

`SINSTR` is an alias of `STEPINSTR` (see page 2-273).

2.3.117 SM

`SM` is an alias of `SETMEM` (see page 2-257).

2.3.118 SOINSTR

`SOINSTR` is an alias of `STEPOINSTR` (see page 2-278).

2.3.119 SOVERLINE

`SOVERLINE` is an alias of `STEPO` (see page 2-280).

2.3.120 SR

`SR` is an alias of `SETREG` (see page 2-259).

2.3.121 STDIOLOG

The `STDIOLOG` command records the messages that are sent to STDIO.

Syntax

`STDIOLOG [/A] [{OFF | ON="`*`filename`*`"}]`

where:

`/A`	Specifies that new records are to be added to any that already exist in the specified file.
`OFF`	Closes the log file and stops collecting information. This is the default.
`ON`	Starts writing information to the log file.
`filename`	Specifies the name of the log file. Quotation marks are optional, but see *Rules for specifying filenames* on page 2-167 for details on how to specify filenames that include a path.

Description

This command records the messages that are sent to STDIO. It does not record any responses you give to prompts.

——— **Note** ———

If you use this command in the **Cmd** tab of the Output pane, the messages are the same as those displayed in the **StdIO** tab of the Output pane.

———————————

If the specified file exists and you do not specify the `/A` parameter, the existing contents of the file are overwritten and lost. A window number (`20`) is associated with the STDIO so that text can be written to it using the commands and macros that support window numbers. See *Window and file numbers* on page 1-5 for details.

Using `STDIOLOG` with no parameters shows the current log file, if any. STDIO output is recorded in the log file until the `STDIOLOG OFF` command is issued.

The `STDIOLOG` command runs asynchronously unless in a macro.

Example

The following examples show how to use `STDIOLOG`:

`STDIOLOG ON='c:\temp\stdiolog.txt'`

Start logging output to the file `c:\temp\stdiolog.txt`, overwriting any existing file of that name.

`STIOLOG /A ON="stdiolog"`

Start logging output to the file `stdiolog.log` in the current directory of the debugger, appending the new log text to the file if it already exists.

`STDIOLOG OFF` Stop logging output.

See also

The following commands provide similar or related functionality:

- *JOURNAL* on page 2-184
- *LOG* on page 2-193
- *VOPEN* on page 2-343.

2.3.122 STEPINSTR

The `STEPINSTR` command executes a specified number of processor instructions.

Note

If a breakpoint is set on an instruction that is encompassed by the `STEPINSTR` command, then the breakpoint is actioned. The breakpoint behavior depends on any condition qualifiers that are assigned. If you do not want the breakpoint to be actioned, then either disable or clear the breakpoint (see *DISABLEBREAK* on page 2-120 or *CLEARBREAK* on page 2-94) before stepping.

Syntax

```
STEPINSTR [value]
```

```
STEPINSTR =starting_address [,value]
```

where:

starting_address

Specifies where execution is to begin. If you do not supply this parameter execution continues from the current PC.

Note

Specifying an address is equivalent to directly modifying the PC. Do not specify a starting address unless you are sure of the consequences to the processor and program state.

value Specifies the number of instructions to be executed.

If you do not supply this parameter a single instruction is executed. All instructions, including instructions that fail a conditional execution test, count towards the number of instructions executed.

Description

The `STEPINSTR` command executes a specified number of instructions. If the instructions include procedure calls, these are stepped into.

Note

For some procedure call standards there is code inserted between the call site and the destination of the call by the linker, and this might not have debug information or source code available. If this is the case for your code, a `STEPINSTR` call that stops in this code causes the source window to be blanked.

It is normal to use this instruction in conjunction with the disassembly mode of the source window, selected using the `MODE` command.

Examples

The following examples show how to use `STEPINSTR`:

`stepinstr`

Step the program by one instruction.

`si 5`

Step the program five instructions.

`si =0x8000,5`

Starting at address `0x8000`, step the program five instructions.

Alias

`SINSTR` is an alias of `STEPINSTR`.

See also

The following commands provide similar or related functionality:

- *BEXECUTION* on page 2-37
- *DISASSEMBLE* on page 2-122
- *GO* on page 2-173
- *GOSTEP* on page 2-175
- *MODE* on page 2-203
- *STEPLINE* on page 2-275.

2.3.123 STEPLINE

The `STEPLINE` command executes one or more program statements, and steps into procedure and function calls.

Syntax

```
STEPLINE [value]
```

```
STEPLINE =starting_address [,value]
```

where:

starting_address

Specifies where execution is to begin. If you do not supply this parameter execution continues from the current PC.

Note

Specifying an address is equivalent to directly modifying the PC. Do not specify a starting address unless you are sure of the consequences to the processor and program state.

value Specifies the number of lines of source code to be executed.

If you do not supply this parameter a single statement or source line is executed. All lines that contain executable code, including those in called functions, count towards the number of lines executed.

Description

The `STEPLINE` command executes one or more source program units. If the debug information in the executable:

- describes the boundaries of program statements, then `STEPLINE` steps by program statement
- describes the source file line for each machine instruction, then `STEPLINE` steps by source line
- describes only the external functions in the code, then `STEPLINE` steps by machine instruction.

`STEPLINE` steps into procedure or function calls. When line or statement debug information is available, the transition from the call site to the first executable statement of the called code counts as one step. If source debug information is available for some but not all of the functions in the program, `STEPLINE` steps to the next source line, whether this is within a called function, for example, from program entry-point to main(), or outside of the current function, for example from an assembler library routine PC to an enclosing source function.

If the step starts in the middle of a statement (for example, because you have used `STEPINSTR`) a single step takes you to the start of the next statement.

If you compile high level language code with debug information and with optimization enabled, for example using `armcc -g -O1`, it is possible that:

- source code is not executed in the order it appears in the source file
- some source program statements are not executed because the optimizer has deduced they are redundant
- some source program statements appear to be not executed because the optimizer has indivisibly combined them with other statements
- statements are executed fewer times than you expect
- it might not be possible to breakpoint or step some statements, because the machine instructions are shared with other source code.

These, and other effects, are the normal consequences of compiler optimization.

For assembler source files assembled with debug information, a single assembly statement consists of;

- an explicitly written assembly instruction
- an assembler pseudo-operation resulting in machine instructions, even if several instructions are generated, for example an ARM `ADR` instruction
- a call of an assembler macro that generates machine instructions.

It is normal to use this instruction in conjunction with the disassembly mode of the source window, selected using the `MODE` command.

Examples

The following examples show how to use `STEPLINE`:

`stepline`

Step the program by one statement.

`stepline 5`

Step the program five statements.

```
s =0x8000,5
```

Starting at address `0x8000`, step the program five statements.

See also

The following commands provide similar or related functionality:

- *BEXECUTION* on page 2-37
- *DISASSEMBLE* on page 2-122
- *GO* on page 2-173
- *GOSTEP* on page 2-175
- *MODE* on page 2-203
- *STEPINSTR* on page 2-273.

2.3.124 STEPOINSTR

The `STEPOINSTR` command executes a specified number of instructions, and completely executes program calls.

Note

If a breakpoint is set on an instruction that is encompassed by the `STEPOINSTR` command, then the breakpoint is actioned. The breakpoint behavior depends on any condition qualifiers that are assigned. If you do not want the breakpoint to be actioned, then either disable or clear the breakpoint (see *DISABLEBREAK* on page 2-120 or *CLEARBREAK* on page 2-94) before stepping.

Syntax

```
STEPOINSTR [value]
```

```
STEPOINSTR =starting_address [,value]
```

where:

starting_address

Specifies where execution is to begin. If you do not supply this parameter execution continues from the current PC.

Note

Specifying an address is equivalent to directly modifying the PC. Do not specify a starting address unless you are sure of the consequences to the processor and program state.

value Specifies the number of instructions to be executed.

If you do not supply this parameter a single instruction is executed. All instructions in the current function, including instructions that fail a conditional execution test, count towards the number of instructions executed. Function calls count as one instruction.

Description

The `STEPOINSTR` command executes a specified number of instructions. If the instructions include procedure calls, these are stepped over, counting as only one instruction.

It is normal to use this instruction in conjunction with the disassembly mode of the source window, selected using the `MODE` command.

Examples

The following examples show how to use `STEPOINSTR`:

`stepoinstr`

Step the program by one instruction.

`stepoinstr 5`

Step the program five instructions.

`soi =0x8000,5`

Starting at address `0x8000`, step the program five instructions, counting a subroutine call as one instruction.

Alias

`SOINSTR` is an alias of `STEPOINSTR`.

See also

The following commands provide similar or related functionality:

- *BEXECUTION* on page 2-37
- *DISASSEMBLE* on page 2-122
- *GO* on page 2-173
- *GOSTEP* on page 2-175
- *MODE* on page 2-203
- *STEPINSTR* on page 2-273
- *STEPLINE* on page 2-275
- *STEPO* on page 2-280.

2.3.125 STEPO

The `STEPO` command executes a specified number of lines, and completely executes functions.

Syntax

```
STEPO [={starting_address [,value] | value}]
```

where:

starting_address
Specifies where execution is to begin. If you do not supply this parameter execution begins at the address currently defined by the PC.

value Specifies the number of lines of source code to be executed. If you do not supply this parameter a single line is executed. All lines in the current program count towards the number of lines executed. A call to a function causes the whole of the function to be executed, and counts as one line.

Description

The `STEPO` command executes one or more source program units. If the debug information in the executable:

- describes the boundaries of program statements, then `STEPO` steps by program statement
- describes the source file line for each machine instruction, then `STEPO` steps by source line
- describes only the function entry points in the code, then `STEPO` steps by machine instruction.

If a statement calls one or more procedures or functions, they are all executed to completion as part of the execution of the statement.

If the step starts in the middle of a statement (for example, because you have used `STEPINSTR`) a single step takes you to the start of the next statement.

If you compile high level language code with debug information and with optimization enabled, for example using `armcc -g -O1`, it is possible that:

- source code is not executed in the order it appears in the source file
- some source program statements are not executed because the optimizer has deduced they are redundant

- some source program statements appear to be not executed because the optimizer has indivisibly combined them with other statements
- statements are executed fewer times than you expect
- it might not be possible to breakpoint or step some statements, because the machine instructions are shared with other source code.

These, and other effects, are the normal consequences of compiler optimization.

For assembler source files assembled with debug information, a single assembly statement consists of;

- an explicitly written assembly instruction
- an assembler pseudo-operation resulting in machine instructions, even if several instructions are generated, for example an ARM `ADR` instruction
- a call of an assembler macro that generates machine instructions.

It is normal to use this instruction in conjunction with the disassembly mode of the source window, selected using the `MODE` command.

Alias

`SO` is an alias of `STEPO`.

Examples

The following examples show how to use `STEPO`:

`stepo`	Step the program by one statement.
`so 5`	Step the program five statements.
`so =0x8000,5`	Starting at address `0x8000`, step the program five statements.

See also

The following commands provide similar or related functionality:

- *BEXECUTION* on page 2-37
- *DISASSEMBLE* on page 2-122
- *GO* on page 2-173
- *GOSTEP* on page 2-175
- *MODE* on page 2-203
- *STEPINSTR* on page 2-273

- *STEPLINE* on page 2-275
- *STEPOINSTR* on page 2-278.

2.3.126 STOP

The `STOP` command stops target program execution, or a specified thread when the processor is running in RSD mode.

Syntax

`STOP [=`*`value`*`]`

where:

`value`	Identifies the thread.

Description

The behavior of the `STOP` command depends on the whether your program is running on a non RTOS-aware or RTOS-aware connection.

Using the STOP command on non RTOS-aware connections

The `STOP` command stops the whole processor.

Using the STOP command on RTOS-aware connections

The behavior of the `STOP` command depends on whether the processor is running in HSD or RSD mode:

- If the processor is running in HSD mode, the command stops the whole processor.
- If the processor is running in RSD mode, and you use the `STOP` command without specifying a thread, the behavior depends on the RTOS `System_Stop` setting in the `Advanced_Information` block for the connection (see the chapter that describes RTOS support in the *RealView Developer Kit v2.2 Extensions User Guide*). If it is set to **Prompt**, you are prompted to continue with the request:
  - **Yes** stops the processor, and the processor falls back to HSD mode.
  - **No** cancels the stop request, and the processor continues to run.
- If the processor is running in RSD mode, then when you are working on a multithreaded image in an unattached Code window, the `STOP` command acts on the current thread. If you are using an attached window, the `STOP` instruction stops the thread that the window is attached to.

  The stopping of threads is accomplished by the Debug Agent using the associated OS service.

If you are working with RTOS images, see the chapters that describe RTOS support and working with multiple target connections in the *RealView Developer Kit v2.2 Extensions User Guide*.

Examples

The following examples show how to use `STOP`:

`stop` Stops the processor.

`stop = thread_4`

Stops the specified thread in RSD.

`stop = 0x39d8`

Stops the thread specified by the TCB address in RSD.

See also

The following commands provide similar or related functionality:

- *AOS_resource-list* on page 2-32
- *BREAKINSTRUCTION* on page 2-61
- *DOS_resource-list* on page 2-129
- *GO* on page 2-173
- *HALT* on page 2-177
- *OSCTRL* on page 2-214

2.3.127 TEST

The TEST command reads target memory to verify that specified values exist throughout the specified memory area.

Syntax

```
TEST [{/B|/H|/W|/8|/16|/32}] [/R] address_range [={expression | stringexpr}]
```

where:

/B, , /8	Sets the expression size to byte (8 bits). This is the default setting.
/H, , /16	Sets the expression size to halfword (16 bits).
/W, , /32	Sets the expression size to word (32 bits).
/R	Continues to test for the specified expression, displaying each match until the end of the block or until the stop button **Cancel** is pressed.
address_range	Specifies the range of addresses to be tested. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.
expression	Specifies a value to check against the contents of memory.
stringexpr	Specifies a string pattern to check against the contents of memory. The debugger tests the memory area to verify that it is filled with those values in the pattern of the string. An expression string is a list of values separated by commas and can include ASCII characters enclosed in quotation marks. All expressions in an expression string are padded or truncated to the size specified by the size qualifiers if they do not fit the specified size evenly.

Description

The TEST command examines target memory to verify that specified values exist throughout the specified memory area. Unless you use the /R qualifier, Testing stops when a mismatch is found. The debugger always displays any mismatched address and value.

Subsequent TEST commands issued without parameters cause the debugger to continue testing through the address range originally specified, beginning with the last address that did not match.

The TEST command runs synchronously.

Examples

The following examples show how to use `TEST`:

```
test/8 0x8000..0x9000 =0
```

Find the address of the first non-zero byte in the 4KB page from `0x8000`.

```
test/r/16 0x10000..0x20000 =0xFFFF
```

Find and display the addresses of any half-word in the address range that is not `0xFFFF`. This might be useful to find out which regions of a Flash memory device are programmed.

See also

The following command provides similar or related functionality:

- *SETMEM* on page 2-257.

2.3.128 THREAD

The `THREAD` command sets the specified thread to be the current thread.

Syntax

```
THREAD [{,next|,default}]
```

`THREAD [=`*`thread`*`]`

where:

`next`	Change the current thread to be the next one in the list of threads.
`default`	Ensures that there is a valid current thread.
`thread`	Define the thread that is to become the current thread. You can use the thread name or the thread ID.

Description

The `THREAD` command sets the specified thread to be the current thread.

The current thread is normally set by whichever thread stops last. This command enables you to specify a thread that is to be the current thread. By default, all actions apply to the current board, process, and thread.

Examples

The following examples show how to use `THREAD`:

```
thread,next
```

Change the current thread to the next thread.

```
thread =thread_2
```

Change the current thread to the thread named `thread_2`.

```
thread =0x13dac
```

Change the current thread to the thread with an ID of `0x13dac`.

See also

The following commands provide similar or related functionality:

- *RUN* on page 2-250.

2.3.129 TRACE

The `TRACE` command provides a quick method of enabling or disabling tracing during program execution. The tracepoints you set with this command are unconditional.

Syntax

```
TRACE  location
```

```
TRACE [{{,endpoint}|{,prompt}|{,trigger}}]  location
```

```
TRACE  ,range  startlocation..endlocation
```

```
TRACE  ,data  startlocation..endlocation
```

where:

location	A program source location, specified symbolically or numerically.
startlocation	The start of a program source range, which must be at a lower address than that specified by *endlocation*.
endlocation	The end of a program source range, which must be at a higher address than that specified by *startlocation*.

Description

The `TRACE` command enables you to set trace trigger, start points, and end points in the program. This enables you to switch tracing on or off at specific addresses during program execution (see *Trace control during program execution*). The tracepoints you set are unconditional tracepoints. To set more complex tracepoints, use the `TRACEDATAACCESS`, `TRACEDATAREAD`, `TRACEDATAWRITE`, `TRACEEXTCOND`, `TRACEINSTREXEC`, or `TRACEINSTRFETCH` command as appropriate.

Trace operation is described in detail in the Trace chapter of the *RealView Debugger v1.8 Extensions User Guide.*

Trace control during program execution

The `endpoint`, `range`, `data`, `prompt`, and `trigger` qualifiers are used to control tracing during program execution. With no qualifier, the `TRACE` command sets a trace start point.

To use these commands, you must specify a program source location, for example a memory address within the program image, or a source module and line number.

The commands are as follows:

`trace` *`location`*

Set a trace start point in the program at address *`location`*.

`trace ,endpoint` *`location`*

Set a trace end point in the program at address *`location`*.

`trace ,trigger` *`location`*

Set a trace trigger in the program at address *`location`*.

`trace ,prompt` *`location`*

Set an unconditional tracepoint in the program at address *`location`*, where the type of tracepoint is selected from a list of supported types presented in a dialog box. See the description on setting unconditional tracepoints in the *RealView Developer Kit v2.2 Extensions User Guide* for more details.

Note

The `prompt` qualifier is not available in the headless debugger.

`trace ,range` *`startlocation..endlocation`*

Set a trace range in the program from address *`startlocation`* to *`endlocation`*, so that instructions at addresses between these points are traced.

`trace ,data` *`startlocation..endlocation`*

Set a trace range in the program from address *`startlocation`* to *`endlocation`*, so that data at addresses between these points are traced.

`trace ,range ,data` *`startlocation..endlocation`*

Set a trace range in the program from address *`startlocation`* to *`endlocation`*, so that instructions executed and data accessed at addresses between these points are traced.

Note

ARM program code often includes literal pools, constants required by the program that cannot be easily included in the instruction opcodes. Literal pool accesses shows up on data tracing, and might quickly fill up the ETM FIFO buffer quickly, depending on the program.

Examples

The following examples show how to use `TRACE`:

`TRACE,prompt \DHRY_1\#78`

Prompts you with a selection of tracepoints that you can set.

`TRACE,range,data 0x80200..0x80400`

Set tracepoints so that data and code accesses between `0x80200-0x80400` are traced, but not accesses at other addresses.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.130 TRACEBUFFER

The `TRACEBUFFER` command manipulates the contents and display of the program execution trace buffer.

Syntax

`TRACEBUFFER ,subcommand [,qualifier] ="text"`

`TRACEBUFFER ,subcommand =value`

`TRACEBUFFER ,subcommand`

where:

subcommand	The possible commands are described in *Description*.
qualifier	The possible qualifiers are described in *Description*.
text	The name of a file or program symbol.
value	A numeric value or range, for example `4` or `5..8`.

Description

The `TRACEBUFFER` command manipulates the program execution and data trace buffer associated with a trace analyzer, enabling you to save, load, find, and filter the data. The actions are differentiated using the *subcommand*, and are described in the section *Subcommands*.

Trace operation is described in detail in the Trace chapter of the *RealView Developer Kit v2.2 Extensions User Guide*.

Subcommands

The possible *subcommands* listed in the syntax are described in the following sections:

- *Loadfile* on page 2-292
- *Savefile* on page 2-292
- *Closefile* on page 2-293
- *Amount* on page 2-293
- *Scaletime* on page 2-294
- *Speed* on page 2-294
- *Find_trigger* on page 2-295
- *Find_position* on page 2-295
- *Find_time* on page 2-295

- *Find_address* on page 2-295
- *Find_data* on page 2-296
- *Find_name* on page 2-296
- *Posfilter* on page 2-296
- *Timefilter* on page 2-296
- *Addressfilter* on page 2-297
- *Namefilter* on page 2-297
- *Percentfilter* on page 2-297
- *Datavaluefilter* on page 2-298
- *Accesstypefilter* on page 2-298
- *Clearfilter* on page 2-298
- *Or_filter* on page 2-298
- *And_filter* on page 2-299
- *Pos_relative* on page 2-299
- *Pos_absolute* on page 2-299
- *Refresh* on page 2-299
- *Gui* on page 2-299.

Loadfile

```
TRACEBUFFER  ,loadfile  ="filename"
```

Load a file into the trace buffer for extra analysis. `filename` is the name of the file to load, and must be quoted.

You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myfiles`, you can specify:

```
tracebuffer,loadfile '$MYPATH\mytrace.dat'
```

Note

The file must have been saved using the `savefile` subcommand with the qualifier `full`, `minimal`, or `profile`, or the GUI equivalent.

Savefile

```
TRACEBUFFER  ,savefile [,ascii|,full|,minimal|,profile] [,append] [,filtered]
="filename"
```

Save the trace buffer to a file, where:

`ascii`	Save lines of text as displayed in the current tab of the Analysis window. This format cannot be loaded into the RealView Debugger analysis window.
`full`	Save the whole trace buffer as a binary file in an RealView Debugger internal format.
`minimal`	Save only timing, address, and access type data from the trace buffer as a binary file in a RealView Debugger internal format. The files created are much smaller than the `full` format, but some information is lost.
`profile`	Save only execution profile data from the trace buffer as a binary file in a RealView Debugger internal format. The files created are smaller than the `minimal` format, but only include enough information to display execution profiles.
`append`	Append the new trace data to an existing file. Do not append data in one format to files in a different format.
`filtered`	Apply the selected display filters when saving trace data. If not specified, the entire trace buffer is saved, regardless of selected display filters.
`filename`	The name of the file to write the data to. If the `full` argument is specified, the filename extension is ignored. If the `full` argument is not specified, then the filename must use a known extension (`.trc`, `.trm`, `.trp`, or `.txt`). You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myfiles`, you can specify: `tracebuffer,savefile '$MYPATH\mytrace.dat'`

Closefile

```
TRACEBUFFER  ,closefile
```

Unload the data from the last file loaded with `loadfile` and clear the Analysis window.

Amount

```
TRACEBUFFER  ,amount  =size
```

This subcommand is deprecated. Specify the number of captured trace records to read from the trace buffer. There is a default value that normally corresponds to the entire trace buffer. Set this if you do not require analysis of all of the captured trace buffer.

The value of *size* is one of:

`0`	The default buffer size. Normally this is the whole buffer, but see your analyzer documentation for full details.
n	The maximum number of records to read.
n..m	The range of records to read, with `0` being the trigger record, if any, and the start of the buffer point if not triggered. If you have a trigger record, you can use negative values to reference records before the trigger. For example, if a trigger is specified then `10..200` means read 190 records starting 10 records after the analyzer triggered. If no trigger is specified, the same string, `10..200`, means to read the 190 records starting 10 records into the buffer. To read the records around the trigger position in the buffer, you can specify `-20..20`.

Scaletime

TRACEBUFFER ,scaletime =*scale*

Set the units for time values displayed in the Analysis window, where *scale* is:

`0`	The default units
`1`	Picoseconds (10^{-12} seconds)
`2`	Nanoseconds (10^{-9} seconds)
`3`	Microseconds (10^{-6} seconds)
`4`	Milliseconds (10^{-3} seconds)
`5`	Seconds
`6`	Cycles.

For ARM ETM, the default units are nanoseconds, and you cannot use scale 6, cycles.

Speed

TRACEBUFFER ,speed =*mhz*

Set the speed of the target processor clock for use in cycle-to-time conversions, where *mhz* is the clock frequency in MHz. The default value is 20MHz. For example:

```
tracebuffer,speed=40
```

sets the speed to 40MHz, so that a period of 400 cycles is considered to take 400/40E6 seconds, or 10 microseconds.

Find_trigger

`TRACEBUFFER ,find_trigger`

Searches for the trigger position in the trace buffer. If found, the item is selected and the Analysis window display is centered on it. There are no arguments.

Find_position

`TRACEBUFFER ,find_position =position`

Searches for the indicated position or set of positions in the trace buffer, where `position` is an integer or range:

n	The position to find.
n..m	Find the first in a range of positions from *n* to *m* inclusive.
n..+o	Find the first in a range of positions from *n* to *n+o* inclusive.

The values *n* and *m* can be negative if a trigger is defined. If any of the positions is found, the first is selected and the Analysis window display is centered on it.

Find_time

`TRACEBUFFER ,find_time =time`

Searches for the indicated time or range of times in the trace buffer, where `time` is an integer or range:

n	The time to find.
n..m	Find the first in a range of times from *n* to *m* inclusive.
n..+o	Find the first in a range of times from *n* to *n+o* inclusive.

The values *n* and *m* can be negative if a trigger is defined. If any of the times are found, the first is selected and the Analysis window display is centered on it.

Find_address

`TRACEBUFFER ,find_address =address`

Searches for the indicated address or set of positions in the trace buffer, where `address` is an integer or range:

n	The address to find.
n..m	Find the first in a range of addresses from *n* to *m* inclusive.
n..+o	Find the first in a range of addresses from *n* to *n+o* inclusive.

If any of the addresses are found, the first is selected and the Analysis window display is centered on it.

Find_data

`TRACEBUFFER ,find_data =dbval`

Searches for the indicated data bus value or set of values in the trace buffer, where `dbval` is an integer or range:

n	The data bus value to find.
n..m	Find the first in a range of data bus values from *n* to *m* inclusive.
n..+o	Find the first in a range of data bus values from *n* to *n+o* inclusive.

The values *n* and *m* can be negative. If any of the values are found, the first is selected and the Analysis window display is centered on it.

Find_name

`TRACEBUFFER ,find_name ="text"`

Searches for the supplied *text*. The search is based on a textual search of the information in the Symbolic column of the analysis window. If found, the record is selected and the Analysis window display is centered on it.

Posfilter

`TRACEBUFFER ,posfilter =position`

Restricts the trace buffer information displayed in the Analysis window based on a positions or set of positions, where *position* is an integer or range:

n	The position to display.
n..m	Display the range of positions from *n* to *m* inclusive. *n* can be negative.
n..+o	Display the range of positions from *n* to *n+o* inclusive.

The values *n* and *m* can be negative if a trigger is defined. Positions are displayed in the `Elem` column of the Analysis window.

Applying a filter to the trace buffer does not lose information unless you save the trace with the `filtered` qualifier. See *Savefile* on page 2-292 for more information.

Timefilter

`TRACEBUFFER ,timefilter =time`

Restricts the trace buffer information displayed in the Analysis window based on a time or range of times in the current time scale units, where *time* is an integer or range:

n	The time to display.
n..m	Display the range of times from *n* to *m* inclusive.
n..+o	Display the range of times from *n* to *n+o* inclusive.

The values *n* and *m* can be negative if a trigger is defined. You can use cycle numbers instead of time values. Applying a filter to the trace buffer does not lose information unless you save the trace with the `filtered` qualifier. See *Savefile* on page 2-292 for more information.

Addressfilter

```
TRACEBUFFER  ,addrfilter  =address
```

```
TRACEBUFFER  ,addressfilter  =address
```

Restricts the trace buffer information displayed in the Analysis window based on an address or range of addresses, where *address* is an integer or range:

n	The address to display.
n..m	Display the range of addresses from *n* to *m* inclusive.
n..+o	Display the range of addresses from *n* to *n+o* inclusive.

You cannot specify addresses symbolically with `addressfilter`. Use `namefilter` instead.

Applying a filter to the trace buffer does not lose information unless you save the trace with the `filtered` qualifier. See *Savefile* on page 2-292 for more information.

Namefilter

```
TRACEBUFFER  ,namefilter  ="name"
```

Restricts the trace buffer information displayed in the Analysis window based on a symbolic name, where *name* is a single string. The symbol names used by this filter are displayed in the Symbolic column of the Analysis window.

Applying a filter to the trace buffer does not lose information unless you save the trace with the `filtered` qualifier. See *Savefile* on page 2-292 for more information.

Percentfilter

```
TRACEBUFFER  ,percentfilter  =percent
```

Restricts the trace buffer information displayed in the Analysis window based on an percentage of the buffer, where *percent* is an integer or range:

n	The percentage to display.
n..m	Display the range of percentages from *n* to *m* inclusive.
n..+o	Display the range of percentages from *n* to *n+o* inclusive.

Applying a filter to the trace buffer does not lose information unless you save the trace with the `filtered` qualifier. See *Savefile* on page 2-292 for more information.

Datavaluefilter

```
TRACEBUFFER  ,dvalfilter  =value
```

```
TRACEBUFFER  ,datavaluefilter  =value
```

Restricts the trace buffer information displayed in the Analysis window based on an data value or range of values, where `value` is an integer or range:

n	The value to display.
n..m	Display the range of value from *n* to *m* inclusive.
n..+o	Display the range of value from *n* to *n+o* inclusive.

Applying a filter to the trace buffer does not lose information unless you save the trace with the `filtered` qualifier. See *Savefile* on page 2-292 for more information.

Accesstypefilter

```
TRACEBUFFER  ,typefilter  =mask
```

```
TRACEBUFFER  ,accesstypefilter  =mask
```

Restricts the trace buffer information displayed in the Analysis window based on an access type, where `mask` is a bitwise-OR of the following values:

`0x001`	Code access.
`0x002`	Data access.
`0x004`	Instruction prefetch.
`0x008`	DMA.
`0x010`	Interrupt.
`0x020`	Bus transaction.
`0x040`	Probe collection.
`0x080`	Pin or signal change.
`0x100`	Non-trace error.

Applying a filter to the trace buffer does not lose information unless you save the trace with the `filtered` qualifier. See *Savefile* on page 2-292 for more information.

Clearfilter

```
TRACEBUFFER  ,clearfilter
```

Remove any and all of the filters applied to the trace buffer, so that the Analysis window displays all the collected trace information.

Or_filter

```
TRACEBUFFER  ,or_filter
```

Specifies that, if multiple types of filter are applied to the trace buffer, that the trace data is displayed if any of the filters display it. That is, the display is the union of all the filters. This is the initial state and you can change it using `and_filter`. Specifying `or_filter` overrides a previously active `and_filter` setting, and the change is applied to the Analysis window immediately.

And_filter

```
TRACEBUFFER  ,and_filter
```

Specifies that, if multiple types of filter are applied to the trace buffer, that the trace data is displayed only if all of the filters display it. That is, the display is the intersection of all the filters. Specifying `and_filter` overrides a previously active `or_filter` setting and the change is applied to the Analysis window immediately.

Pos_relative

```
TRACEBUFFER  ,pos_relative
```

Specifies that the element (position) numbering used in the `Elem` column of the Analysis window is relative to the trigger position, so that the trigger record is numbered `0`, the record before (in time) the trigger is `-1`, and the record after is `1`.

Pos_absolute

```
TRACEBUFFER  ,pos_absolute
```

Specifies that the element (position) numbering used in the `Elem` column of the Analysis window is absolute, so that the record captured first is numbered `0`, and records captured later are numbered in increasing sequence.

You cannot use this mode with the ARM ETM because records are always relative to a trigger.

Refresh

```
TRACEBUFFER  ,refresh
```

This option refreshes the trace display.

Gui

```
TRACEBUFFER  ,gui
```

This option modifies the action of the other commands. It specifies that the `TRACEBUFFER` command was initiated from the GUI, and that messages must be displayed using dialogs rather than text in the command window.

Note

This option has no effect in the headless debugger.

Examples

The following examples show how to use `TRACEBUFFER`:

`TRACEBUFFER,timefilter 49.9..50.1`

Set a filter that displays in the Analysis window only trace records captured 0.1 time unit before and after 50 time units. You set time units with `scaletime`.

`TRACEBUFFER,savefile,full ="tracerun.trc"`

Save the whole of the current trace buffer, because no filtering is applied, to a file in the current directory called `tracerun.trc`.

`TRACEBUFFER,find_name ="main"`

Search through the Analysis window for the first occurrence of the text `main`, and display it.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.131 TRACEDATAACCESS

The TRACEDATAACCESS command sets a trace point on data accesses, that is, either reads or writes.

——— **Note** ———

This command is valid only for ETM-based hardware targets.

———————————

Syntax

```
TRACEDATAACCESS  [,qualifier...]  {address |  address_range}
```

where:

qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers*.

address Specifies the address at which the tracepoint is placed.

address_range

Specifies the address range for the tracepoint. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

This command sets a tracepoint at the address or address range you specify that triggers when an instruction access at the indicated address accesses data from memory.

The tracepoint type is by default to trigger, that is, start collecting trace information into the trace buffer. You can modify the action using the hw_out: qualifier to, for example, stop tracing.

For more information about tracepoints and the way you access the ETM, see the *Embedded Trace Macrocell Specification* and the chapter that describes tracing in *RealView Developer Kit v2.2 Extensions User Guide*.

List of qualifiers

The command qualifiers are as follows, but not all qualifiers are available for all of the supported trace targets:

gui If an error occurs when executing the command or when the tracepoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

Note

This qualifier has no effect in the headless debugger.

`hw_ahigh:(n)`	Specifies the high address for an address-range tracepoint. The low address is specified by the standard tracepoint address. For example, this command sets a tracepoint that triggers when a data access is made by any instruction in the address range from `0x1000` to `0x1200`: `TRCDA,hw_ahigh:0x1200 0x1000`
`hw_dvalue:(n)`	Specifies a data value to be compared to values transmitted on the processor data bus. For example, this command sets a tracepoint that triggers for a data read or a data write of the data value `0x400` at an instruction at address `0x1FA00`: `TRCDA,hw_dvalue:0x400 0x1FA00`
`hw_dhigh:(n)`	Specifies the high data value for a data-range tracepoint. The low data value is specified by the `hw_dvalue` qualifier. For example, this command sets a tracepoint that triggers for any data read or a data write of the data value between `0x00-0x18` at an instruction at address `0x1000`: `TRCDA,hw_dvalue:0x0,hw_dhigh:0x18 0x1000`
`hw_dmask:(n)`	Specifies the data value mask value used for in comparisons with a data-value tracepoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the tracepoint to trigger. For example, this command sets a tracepoint that triggers for any data read or a data write of the data value between `0x400-0x4FF` at an instruction at address `0x1000`: `TRCDA,hw_dvalue:0x400,hw_dmask:0xF00 0x1000`
`hw_passcount:(n)`	Specifies the number of times that the specified condition has to occur to trigger the tracepoint. You can use this option to set up and use the ARM ETM counter hardware, if the ETM has counters exists and there is one available for use. ETM counters are 32 bits.

`hw_and:{[then-]id}` Perform an *and* or an *and-then* conjunction with an existing tracepoint. For example, `hw_and:h2`, or `hw_and:"then-h2"`, where *2* is the tracepoint id of another tracepoint.

In the *and* form, the conditions associated with both tracepoints are chained together, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the *and-then* form, when the condition for the first tracepoint is met, the second tracepoint is enabled. When the second tracepoint condition is matched, even if the first condition no longer matches, the actions associated are performed.

The *id* is one of:

- the tracepoint list index of an existing of tracepoint, and has the format `h`*index*, for example `h1`
- `prev` for the last tracepoint specified for this connection
- `next` for the target of this condition.

For example, these two commands set a tracepoint that triggers the trace buffer when a data access in code at line `582` of `modify.c` is followed by another data access at line `379` of `access.c`:

```
TRCDA \MODIFY_1\#582
TRCDA,hw_and:then-1 \ACCESS_1\#379
```

`hw_in:{s}` Input trigger tests. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"Size of Data Access=s"`

For data comparisons, check the data access size against the specified value, and return True if it matches, where *s* is:

`Any`	Any access size.
`Halfword`	16-bit accesses.
`Word`	32-bit accesses.

For example, this command sets a tracepoint that triggers for any 16-bit data read or a write that occurs in the program code between `0x1E000-0x1FF00`:

```
TRCDA,hw_in:"Size of Data Access=Halfword" 0x1E00..0x1FF00
```

`hw_out:{s}`	Output trigger tests. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive form is defined:

`"Tracepoint Type=s"`

Specify the trace action for this command, where *s* is:

`Trigger`	Sets a trigger point.
`Start Tracing`	Sets a trace start point.
`Stop Tracing`	Sets a trace stop point.
`Trace Instr`	Sets an instruction-only trace range.
`Trace Instr and Data`	Sets an instruction and data trace range.

For example, this command sets a tracepoint that traces all instructions executed between program code addresses `0x1E000-0x1FF00`, but does not trace instructions outside this range:

```
TRCDA,hw_out:"Tracepoint Type=Trace Instr" 0x1E00..0x1FF00
```

`hw_not:{s}` Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *s* can be set to:

`addr`	Invert the tracepoint address value.
`data`	Invert the tracepoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to trace when a data value does not match a mask, you can write:

```
TRCDA,hw_not:data,hw_dmask:0x00FF ...
```

The trace commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
TRCDA,hw_not:addr 0x10040..0x10060
```

This means to trace execution at addresses other than the range `0x10040-0x10060`, that is, exclude this region from the trace.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *or* and *nand-then* conditions.

`modify:(n)` Instead of creating a new tracepoint, modify the tracepoint with tracepoint ID number *n* by replacing the address expression and the qualifiers of the existing tracepoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained tracepoint to a chained tracepoint. However, you can modify a chained tracepoint with any other qualifier and also change the address expression.

Examples

The following examples show how to use `TRACEDATAACCESS`:

`TRACEDATAACCESS \MATH_1\#449.3`

Set a trace trigger at statement 3 of line 449 in the file `math.c`.

`TRCDA \MAIN_1\#35..\MAIN_1\#63`

Start tracing instructions when a data access in the code between line 35-63 of `main.c` occurs.

`TRCDA,hw_pass:5,hw_out:"Tracepoint Type=Start Tracing" \MAIN_1\#35`

Start tracing when a data access at line 35 of `main.c` occurs.

`TRCDA,hw_out:"Tracepoint Type=Stop Tracing" \GUI_1\#35..\GUI_1\#78`

Stop tracing when any instruction between line 35-78 of `gui.c` accesses data.

Alias

`TRCDACCES` is an alias of `TRACEDATAACCESS`.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318

- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.132 TRACEDATAREAD

The `TRACEDATAREAD` command enables you to set a tracepoint on data reads.

Note

This command is valid only for ETM-based hardware targets.

Syntax

```
TRACEDATAREAD [,qualifier...] {address | address_range}
```

where:

qualifier	Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers*.
address	Specifies the address at which the tracepoint is placed.
address_range	Specifies the address range at which the tracepoint is placed. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

This command sets a tracepoint at the address or address range you specify that triggers when an instruction access at the indicated address reads data from memory.

The tracepoint type is by default to trigger, that is, start collecting trace information into the trace buffer. You can modify the action using the `hw_out:` qualifier to, for example, stop tracing.

For more information about tracepoints and the way you access the ETM, see the *Embedded Trace Macrocell Specification* and the chapter that describes tracing in *RealView Developer Kit v2.2 Extensions User Guide*.

List of qualifiers

The command qualifiers are as follows, but not all qualifiers are available for all of the supported trace targets:

`gui`	If an error occurs when executing the command or when the tracepoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

Note

This qualifier has no effect in the headless debugger.

`hw_ahigh:(n)`
: Specifies the high address for an address-range tracepoint. The low address is specified by the standard tracepoint address.

For example, this command sets a tracepoint that triggers when a data read is made by any instruction in the address range from `0x1000` to `0x1200`:

```
TRCDR,hw_ahigh:0x1200 0x1000
```

`hw_dvalue:(n)`
: Specifies a data value to be compared to values transmitted on the processor data bus.

For example, this command sets a tracepoint that triggers for a data read of the data value `0x400` at an instruction at address `0x1FA00`:

```
TRCDR,hw_dvalue:0x400 0x1FA00
```

`hw_dhigh:(n)`
: Specifies the high data value for a data-range tracepoint. The low data value is specified by the `hw_dvalue` qualifier.

For example, this command sets a tracepoint that triggers for any data read of the data value between `0x00-0x18` at an instruction at address `0x1000`:

```
TRCDR,hw_dvalue:0x0,hw_dhigh:0x18 0x1000
```

`hw_dmask:(n)`
: Specifies the data value mask value used for in comparisons with a data-value tracepoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the tracepoint to trigger.

For example, this command sets a tracepoint that triggers for any data read of the data value between `0x400-0x4FF` at an instruction at address `0x1000`:

```
TRCDR,hw_dvalue:0x400,hw_dmask:0xF00 0x1000
```

`hw_passcount:(n)`
: Specifies the number of times that the specified condition has to occur to trigger the tracepoint.

You can use this option to set up and use the ARM ETM counter hardware, if the ETM has counters exists and there is one available for use. ETM counters are 32 bits.

`hw_and:{[then-]id}` Perform an *and* or an *and-then* conjunction with an existing tracepoint. For example, `hw_and:h2`, or `hw_and:"then-h2"`, where *2* is the tracepoint id of another tracepoint.

In the *and* form, the conditions associated with both tracepoints are chained together, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the *and-then* form, when the condition for the first tracepoint is met, the second tracepoint is enabled. When the second tracepoint condition is matched, even if the first condition no longer matches, the actions associated are performed.

The *id* is one of:

- the tracepoint list index of an existing of tracepoint, and has the format `h`*index*, for example `h1`
- `prev` for the last tracepoint specified for this connection
- `next` for the target of this condition.

For example, these two commands set a tracepoint that triggers the trace buffer when a data access in code at line `582` of `modify.c` is followed by another data access at line `379` of `access.c`:

```
TRCDR \MODIFY_1\#582
TRCDR,hw_and:then-1 \ACCESS_1\#379
```

`hw_in:{s}` Input trigger tests. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"Size of Data Access=s"`

For data comparisons, check the data access size against the specified value, and return True if it matches, where *s* is:

`Any`	Any access size.
`Halfword`	16-bit accesses.
`Word`	32-bit accesses.

For example, this command sets a tracepoint that triggers for any 16-bit data read that occurs in the program code between `0x1E000-0x1FF00`:

```
TRCDR,hw_in:"Size of Data Access=Halfword" 0x1E00..0x1FF00
```

`hw_out:{s}`	Output trigger tests. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive form is defined:

`"Tracepoint Type=s"`

Specify the trace action for this command, where *s* is:

`Trigger`	Sets a trigger point.
`Start Tracing`	Sets a trace start point.
`Stop Tracing`	Sets a trace stop point.
`Trace Instr`	Sets an instruction-only trace range.
`Trace Instr and Data`	Sets an instruction and data trace range.

For example, this command sets a tracepoint that traces all instructions executed between program code addresses `0x1E000-0x1FF00`, but does not trace instructions outside this range:

```
TRCDR,hw_out:"Tracepoint Type=Trace Instr" 0x1E00..0x1FF00
```

`hw_not:{s}`	Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *s* can be set to:

`addr`	Invert the tracepoint address value.
`data`	Invert the tracepoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to trace when a data value does not match a mask, you can write:

```
TRCDR,hw_not:data,hw_dmask:0x00FF ...
```

The trace commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
TRCDR,hw_not:addr 0x10040..0x10060
```

This means to trace execution at addresses other than the range `0x10040` to `0x10060`, that is, exclude this region from the trace.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *or* and *nand-then* conditions.

`modify:(n)`	Instead of creating a new tracepoint, modify the tracepoint with tracepoint ID number *n* by replacing the address expression and the qualifiers of the existing tracepoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained tracepoint to a chained tracepoint. However, you can modify a chained tracepoint with any other qualifier and also change the address expression.

Examples

The following examples show how to use `TRACEDATAREAD`:

```
TRACEDATAREAD \COMMAND_1\#132
```

Set a trace data read trigger at line 132 in the file `command.c`.

```
TRCDR \MAIN_1\#35..\MAIN_1\#63
```

Start tracing instructions when a data read occurs in the code between line 35-63 of `main.c`.

```
TRCDR,hw_pass:5,hw_out:"Tracepoint Type=Start Tracing" \MAIN_1\#35
```

Start tracing when a data read occurs at line 35 of `main.c`.

```
TRCDR,hw_out:"Tracepoint Type=Stop Tracing" \GUI_1\#35..\GUI_1\#78
```

Stop tracing when any instruction between line 35-78 of `gui.c` reads data.

Alias

`TRCDREAD` is an alias of `TRACEDATAREAD`.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318

- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.133 TRACEDATAWRITE

The `TRACEDATAWRITE` command enables you to set a tracepoint on data reads.

——— **Note** ———

This command is valid only for ETM-based hardware targets.

———————————

Syntax

`TRACEDATAWRITE [,qualifier...] {address | address_range}`

where:

qualifier	Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers*.
address	Specifies the address at which the tracepoint is placed.
address_range	Specifies the address range at which the tracepoint is placed. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

This command sets a tracepoint at the address or address range you specify that triggers when an instruction access at the indicated address writes data to memory.

The tracepoint type is by default to trigger, that is, start collecting trace information into the trace buffer. You can modify the action using the `hw_out:` qualifier to, for example, stop tracing.

For more information about tracepoints and the way you access the ETM, see the *Embedded Trace Macrocell Specification* and the chapter that describes tracing in *RealView Developer Kit v2.2 Extensions User Guide*.

List of qualifiers

The command qualifiers are as follows, but not all qualifiers are available for all of the supported trace targets:

`gui`	If an error occurs when executing the command or when the tracepoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

Note

This qualifier has no effect in the headless debugger.

`hw_ahigh:(n)`	Specifies the high address for an address-range tracepoint. The low address is specified by the standard tracepoint address. For example, this command sets a tracepoint that triggers when data is written by any instruction in the address range from `0x1000` to `0x1200`: `TRCDW,hw_ahigh:0x1200 0x1000`
`hw_dvalue:(n)`	Specifies a data value to be compared to values transmitted on the processor data bus. For example, this command sets a tracepoint that triggers when data value `0x400` is written by an instruction at address `0x1FA00`: `TRCDW,hw_dvalue:0x400 0x1FA00`
`hw_dhigh:(n)`	Specifies the high data value for a data-range tracepoint. The low data value is specified by the `hw_dvalue` qualifier. For example, this command sets a tracepoint that triggers when data value between `0x00-0x18` is written by an instruction at address `0x1000`: `TRCDW,hw_dvalue:0x0,hw_dhigh:0x18 0x1000`
`hw_dmask:(n)`	Specifies the data value mask value used for in comparisons with a data-value tracepoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the tracepoint to trigger. For example, this command sets a tracepoint that triggers when data value between `0x400-0x4FF` is written by an instruction at address `0x1000`: `TRCDW,hw_dvalue:0x400,hw_dmask:0xF00 0x1000`
`hw_passcount:(n)`	Specifies the number of times that the specified condition has to occur to trigger the tracepoint. You can use this option to set up and use the ARM ETM counter hardware, if the ETM has counters exists and there is one available for use. ETM counters are 32 bits.
`hw_and:{[then-]id}`	Perform an *and* or an *and-then* conjunction with an existing tracepoint. For example, `hw_and:h2`, or `hw_and:"then-h2"`, where *2* is the tracepoint id of another tracepoint.

In the *and* form, the conditions associated with both tracepoints are chained together, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the *and-then* form, when the condition for the first tracepoint is met, the second tracepoint is enabled. When the second tracepoint condition is matched, even if the first condition no longer matches, the actions associated are performed.

The `id` is one of:

- the tracepoint list index of an existing of tracepoint, and has the format `hindex`, for example `h1`
- `prev` for the last tracepoint specified for this connection
- `next` for the target of this condition.

For example, these two commands set a tracepoint that triggers the trace buffer when a data write in code at line `582` of `modify.c` is followed by another data write at line `379` of `access.c`:

```
TRCDW \MODIFY_1\#582
TRCDW,hw_and:then-1 \ACCESS_1\#379
```

`hw_in:{s}` Input trigger tests. The string `s` is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"Size of Data Access=s"`

For data comparisons, check the data access size against the specified value, and return True if it matches, where `s` is:

`Any`	Any access size.
`Halfword`	16-bit accesses.
`Word`	32-bit accesses.

For example, this command sets a tracepoint that triggers for any 16-bit data read or a write that occurs in the program code between `0x1E000-0x1FF00`:

```
TRCDW,hw_in:"Size of Data Access=Halfword" 0x1E00..0x1FF00
```

`hw_out:{s}`	Output trigger tests. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive form is defined:

`"Tracepoint Type=s"`

Specify the trace action for this command, where *s* is:

`Trigger`	Sets a trigger point.
`Start Tracing`	Sets a trace start point.
`Stop Tracing`	Sets a trace stop point.
`Trace Instr`	Sets an instruction-only trace range.
`Trace Instr and Data`	Sets an instruction and data trace range.

For example, this command sets a tracepoint that traces all instructions executed between program code addresses `0x1E000-0x1FF00`, but does not trace instructions outside this range:

```
TRCDW,hw_out:"Tracepoint Type=Trace Instr" 0x1E00..0x1FF00
```

`hw_not:{s}`	Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *s* can be set to:

`addr`	Invert the tracepoint address value.
`data`	Invert the tracepoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to trace when a data value does not match a mask, you can write:

```
TRCDW,hw_not:data,hw_dmask:0x00FF ...
```

The trace commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
TRCDW,hw_not:addr 0x10040..0x10060
```

This means to trace execution at addresses other than the range `0x10040` to `0x10060`, that is, exclude this region from the trace.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *or* and *nand-then* conditions.

`modify:(n)`	Instead of creating a new tracepoint, modify the tracepoint with tracepoint ID number *n* by replacing the address expression and the qualifiers of the existing tracepoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained tracepoint to a chained tracepoint. However, you can modify a chained tracepoint with any other qualifier and also change the address expression.

Examples

The following example shows how to use `TRACEDATAWRITE`:

```
TRACEDATAWRITE \MATH_1\#449.3
```

Set a trace data write trigger at statement 3 of line 449 in the file `math.c`.

Alias

`TRCDWRITE` is an alias of `TRACEDATAWRITE`.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.134 TRACEEXTCOND

The `TRACEEXTCOND` command enables you to set a tracepoint that triggers when a specified external condition is enabled.

Note

This command is valid only for ETM-based hardware targets.

Syntax

```
TRACEEXTCOND  [,qualifier...]  {address | address_range}
```

where:

qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers*.

address Specifies the address at which the tracepoint is placed.

address_range

Specifies the address range at which the tracepoint is placed. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

This command sets a tracepoint at the address or address range you specify. The tracepoint triggers when an instruction in the specified address range is executed. By default, the tracepoint type is `Trigger`, that is start collecting trace information into the trace buffer. You can modify the action using the `hw_out:` qualifiers to, for example, stop tracing.

For more information about tracepoints and the way you access the ETM, see the *Embedded Trace Macrocell Specification* and the chapter that describes tracing in *RealView Developer Kit v2.2 Extensions User Guide*.

List of qualifiers

The command qualifiers are as follows, but not all qualifiers are available for all of the supported trace targets:

`gui` If an error occurs when executing the command or when the tracepoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

—— **Note** ——

This qualifier has no effect in the headless debugger.

——

`hw_passcount:(`*`n`*`)`	Specifies the number of times that the specified condition has to occur to trigger the tracepoint. You can use this option to set up and use the ARM ETM counter hardware, if the ETM has counters exists and there is one available for use. ETM counters are 32 bits.

`hw_and:{[then-]`*`id`*`}` Perform an *and* or an *and-then* conjunction with an existing tracepoint. For example, `hw_and:h2`, or `hw_and:"then-h2"`, where *2* is the tracepoint id of another tracepoint.

In the *and* form, the conditions associated with both tracepoints are chained together, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the *and-then* form, when the condition for the first tracepoint is met, the second tracepoint is enabled. When the second tracepoint condition is matched, even if the first condition no longer matches, the actions associated are performed.

The *`id`* is one of:

- the tracepoint list index of an existing of tracepoint, and has the format `h`*`index`*, for example `h1`
- `prev` for the last tracepoint specified for this connection
- `next` for the target of this condition.

`hw_in:{s}`	Input trigger to test for external condition events. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"External Condition=s"`

The tracepoint is activated on the events shown in Table 2-16.

Table 2-16 External condition events

Event	String setting
External inputs 1-4	`ExternalIn1` `ExternalIn2` `ExternalIn3` `ExternalIn4`
EmbeddedICE watchpoints 1-2	`Watchpoint1` `Watchpoint2`
Access to ASIC memory maps 1-16	`ASIC Memmap 1 ... ASIC Memmap 16`

Up to four signals are available. The ASIC manufacturer determines the availability and usage of these output signals. See your ASIC documentation for details.

`hw_out:{s}`	Output trigger tests. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"Tracepoint Type=s"`

Specify the trace action for this command, where *s* is:

`Trigger`	Sets a trigger point.
`Start Tracing`	Sets a trace start point.
`Stop Tracing`	Sets a trace stop point.
`Trace Instr`	Sets an instruction-only trace range.
`Trace Instr and Data`	Sets an instruction and data trace range.

`hw_not:{s}`
: Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *s* can be set to:

`addr`	Invert the tracepoint address value.
`data`	Invert the tracepoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to trace when a data value does not match a mask, you can write:

```
TRACEEXTCOND,hw_not:data,hw_dmask:0x00FF ...
```

The trace commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
TRACEEXTCOND,hw_not:addr 0x10040..0x10060
```

This means to trace execution at addresses other than the range `0x10040` to `0x10060`, that is, exclude this region from the trace.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *or* and *nand-then* conditions.

`modify:(n)`
: Instead of creating a new tracepoint, modify the tracepoint with tracepoint ID number *n* by replacing the address expression and the qualifiers of the existing tracepoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained tracepoint to a chained tracepoint. However, you can modify a chained tracepoint with any other qualifier and also change the address expression.

Examples

The following examples show how to use `TRACEEXTCOND`:

`TRACEEXTCOND \MATH_1\#449.3`

Set a hardware tracepoint at statement 3 of line 449 in the file `math.c`.

`TRACEEXTCOND,hw_pass:(5) \MAIN_1\#35`

Set a hardware tracepoint using an ETM counter to enable tracing the fifth time that execution reaches line 35 of `main.c`.

Alias

TRCEEXTC is an alias of TRACEEXTCOND.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEINSTREXEC* on page 2-323
- *TRACEINSTRFETCH* on page 2-328.

2.3.135 TRACEINSTREXEC

The `TRACEINSTREXEC` command enables you to set a tracepoint on instruction execution.

Syntax

```
TRACEINSTREXEC  [,qualifier...]  {address | address_range}
```

where:

qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers*.

address Specifies the address at which the tracepoint is placed.

address_range

Specifies the address range at which the tracepoint is placed. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

This command sets a tracepoint at the address or address range you specify that triggers when an instruction is executed in the indicated address range.

The tracepoint type is by default to trigger, that is, start collecting trace information into the trace buffer. You can modify the action using the `hw_out:` qualifier to, for example, stop tracing.

For more information about tracepoints and the way you access the ETM, see the *Embedded Trace Macrocell Specification* and the chapter that describes tracing in *RealView Developer Kit v2.2 Extensions User Guide*.

List of qualifiers

The command qualifiers are as follows, but not all qualifiers are available for all of the supported trace targets:

`gui` If an error occurs when executing the command or when the tracepoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

Note

This qualifier has no effect in the headless debugger.

`hw_ahigh:(`*n*`)`	Specifies the high address for an address-range tracepoint. The low address is specified by the standard tracepoint address. For example, this command sets a tracepoint that triggers for any address between `0x1000-0x1200`: `TRCIE,hw_ahigh:0x1200 0x1000`
`hw_dvalue:(`*n*`)`	Specifies a data value to be compared to values transmitted on the processor data bus. For example, this command sets a tracepoint that triggers when the instruction opcode `0xEA000040` is executed in code between `0x1FA00-0x1FAFF`: `TRCIE,hw_dvalue:0xEA000040 0x1FA00..0x1FAFF`

`hw_dhigh:(`*n*`)`	Specifies the high data value for a data-range tracepoint. The low data value is specified by the `hw_dvalue` qualifier.
	For example, this command sets a tracepoint that triggers when the instruction opcode between `0xEA000040-0xEA00004F` is executed in code between `0x1FA00-0x1FAFF`:
	`TRCIE,hw_dvalue:0xEA000040,hw_dhigh:0xEA00004F`
	`0x1FA00..0x1FAFF`

`hw_dmask:(`*n*`)`	Specifies the data value mask value for a data-range tracepoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the tracepoint to trigger.
	For example, this command sets a tracepoint that triggers when the an instruction with basic opcode `0xEA000040` but with any value in bits [15:8] is executed in code between `0x1FA00-0x1FAFF`:
	`TRCIE,hw_dvalue:0xEA000040,hw_dmask:0xFFFF00FF`
	`0x1FA00..0x1FAFF`

<u>`hw_pass`</u>`count:(`*n*`)`	Specifies the number of times that the specified condition has to occur to trigger the tracepoint. You can use this option to set up and use the ARM ETM counter hardware, if the ETM has counters exists and there is one available for use. ETM counters are 32 bits.
`hw_and:{[then-]`*id*`}`	Perform an *and* or an *and-then* conjunction with an existing tracepoint. For example, `hw_and:h2`, or `hw_and:"then-h2"`, where *2* is the tracepoint id of another tracepoint. In the *and* form, the conditions associated with both tracepoints are chained together, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the *and-then* form, when the condition for the first tracepoint is met, the second tracepoint is enabled. When the second tracepoint condition is matched, even if the first condition no longer matches, the actions associated are performed.

The `id` is one of:

- the tracepoint list index of an existing of tracepoint, and has the format `h`*`index`*, for example `h1`
- `prev` for the last tracepoint specified for this connection
- `next` for the target of this condition.

`hw_in:{`*`s`*`}` Input trigger tests. The string *`s`* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"Check Condition Code=`*`s`*`"`

For instruction tracepoints, comparisons, check the instruction condition code against the specified value, and return True if it matches, where *`s`* is:

`Pass`	Trace only instructions that are executed.
`Fail`	Trace only instructions that are not executed.

`hw_out:{`*`s`*`}` Output trigger tests. The string *`s`* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"Tracepoint Type=`*`s`*`"`

Specify the trace action for this command, where *`s`* depends on the target connection:

- For an ETM-based hardware target *s* is:

`Trigger`	Sets a trigger point.
`Start Tracing`	Sets a trace start point.
`Stop Tracing`	Sets a trace stop point.
`Trace Instr`	Sets an instruction-only trace range.
`Trace Instr and Data`	Sets an instruction and data trace range.

`hw_not:{s}`
: Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *`s`* can be set to:

`addr`	Invert the tracepoint address value.
`data`	Invert the tracepoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to trace when a data value does not match a mask, you can write:

```
TRCIE,hw_not:data,hw_dmask:0x00FF ...
```

The trace commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
TRCIE,hw_not:addr 0x10040..0x10060
```

This means to trace execution at addresses other than the range `0x10040` to `0x10060`, that is, exclude this region from the trace.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *or* and *nand-then* conditions.

`modify:(n)`
: Instead of creating a new tracepoint, modify the tracepoint with tracepoint ID number *`n`* by replacing the address expression and the qualifiers of the existing tracepoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained tracepoint to a chained tracepoint. However, you can modify a chained tracepoint with any other qualifier and also change the address expression.

Examples

The following examples show how to use `TRACEINSTREXEC`:

```
TRACEINSTREXEC \MATH_1\#449.3
```

Set a hardware tracepoint at statement 3 of line 449 in the file `math.c`.

```
TRCIE,hw_pass:(5) \MAIN_1\#35
```

Set a hardware tracepoint using an ETM counter to enable tracing the fifth time that execution reaches line 35 of `main.c`.

Alias

TRCIEXEC is an alias of TRACEINSTREXEC.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTBREAK* on page 2-135
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291
- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTRFETCH* on page 2-328.

2.3.136 TRACEINSTRFETCH

The `TRACEINSTRFETCH` command enables you to set a tracepoint on instruction fetch from memory.

Note

This command is valid only for ETM-based hardware targets.

Syntax

```
TRACEINSTRFETCH [,qualifier...] {address | address_range}
```

where:

qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in *List of qualifiers* on page 2-329.

address Specifies the address at which the tracepoint is placed.

address_range

Specifies the address range at which the tracepoint is placed. See *Specifying address ranges* on page 2-2 for details on how to specify an address range.

Description

This command sets a tracepoint at the address or address range you specify that triggers when an instruction opcode is fetched from memory in the indicated address range.

Note

Use this type of tracepoint with care, because not all instructions that are fetched are executed, and because the fetch from memory occurs several cycles before execution and possibly not in execution order.

The tracepoint type is by default to trigger, that is, start collecting trace information into the trace buffer. You can modify the action using the `hw_out:` qualifier to, for example, stop tracing.

For more information about tracepoints and the way you access the ETM, see the *Embedded Trace Macrocell Specification* and the chapter that describes tracing in *RealView Developer Kit v2.2 Extensions User Guide*.

List of qualifiers

The command qualifiers are as follows, but not all qualifiers are available for all of the supported trace targets:

`gui`
: If an error occurs when executing the command or when the tracepoint is triggered, the GUI is used to report it. Otherwise, the error is reported to the command pane.

  Note

  This qualifier has no effect in the headless debugger.

`hw_ahigh:(n)`
: Specifies the high address for an address-range tracepoint. The low address is specified by the standard tracepoint address.

  For example, this command sets a tracepoint that triggers for any address between `0x1000-0x1200`:

```
TRCIF,hw_ahigh:0x1200 0x1000
```

`hw_dvalue:(n)`
: Specifies a data value to be compared to values transmitted on the processor data bus.

  For example, this command sets a tracepoint that triggers when the instruction opcode `0xEA000040` is fetched in code between `0x1FA00-0x1FAFF`:

```
TRCIF,hw_dvalue:0xEA000040 0x1FA00..0x1FAFF
```

`hw_dhigh:(n)`
: Specifies the high data value for a data-range tracepoint. The low data value is specified by the `hw_dvalue` qualifier.

  For example, this command sets a tracepoint that triggers when the instruction opcode between `0xEA000040-0xEA00004F` is fetched in code between `0x1FA00-0x1FAFF`:

```
TRCIF,hw_dvalue:0xEA000040,hw_dhigh:0xEA00004F
0x1FA00..0x1FAFF
```

`hw_dmask:(n)`
: Specifies the data value mask value for a data-range tracepoint. Data values that match the value specified by the `hw_dvalue` qualifier when masked with this value cause the tracepoint to trigger.

  For example, this command sets a tracepoint that triggers when the an instruction with basic opcode `0x0xEA000040` but with any value in bits 8-15 is fetched in code between `0x1FA00-0x1FAFF`:

```
TRCIF,hw_dvalue:0xEA000040,hw_dmask:0xFFFF00FF
0x1FA00..0x1FAFF
```

`hw_passcount:(n)`	Specifies the number of times that the specified condition has to occur to trigger the tracepoint. You can use this option to set up and use the ARM ETM counter hardware, if the ETM has counters exists and there is one available for use. ETM counters are 32 bits.

`hw_and:{[then-]id}` Perform an *and* or an *and-then* conjunction with an existing tracepoint. For example, `hw_and:h2`, or `hw_and:"then-h2"`, where *2* is the tracepoint id of another tracepoint.

In the *and* form, the conditions associated with both tracepoints are chained together, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the *and-then* form, when the condition for the first tracepoint is met, the second tracepoint is enabled. When the second tracepoint condition is matched, even if the first condition no longer matches, the actions associated are performed.

The*id* is one of:

- the tracepoint list index of an existing of tracepoint, and has the format `h`*`index`*, for example `h1`
- `prev` for the last tracepoint specified for this connection
- `next` for the target of this condition.

`hw_in:{s}` Input trigger tests. The string *s* is specific to the trace connection being used.

`hw_out:{s}` Output trigger tests. The string *s* is specific to the trace connection being used. For the ARM ETM, the following case-sensitive forms are defined:

`"Tracepoint Type=s"`

Specify the trace action for this command, where *s* is:

`Trigger`	Sets a trigger point.
`Start Tracing`	Sets a trace start point.
`Stop Tracing`	Sets a trace stop point.
`Trace Instr`	Sets an instruction-only trace range.
`Trace Instr and Data`	Sets an instruction and data trace range.

`hw_not:{s}`

Use this qualifier to invert the sense of an address, data, or `hw_and` term specified in the same command. The argument *`s`* can be set to:

`addr`	Invert the tracepoint address value.
`data`	Invert the tracepoint value.
`then`	Invert an associated `hw_and:{then}` condition.

For example, to trace when a data value does not match a mask, you can write:

```
TRCIF,hw_not:data,hw_dmask:0x00FF ...
```

The trace commands require an address value, and the `addr` variant of `hw_not` uses this address.

```
TRCIF,hw_not:addr 0x10040..0x10060
```

This means to trace execution at addresses other than the range `0x10040` to `0x10060`, that is, exclude this region from the trace.

The `hw_not:then` variant of the command is used in conjunction with `hw_and` to form *or* and *nand-then* conditions.

`modify:(n)`

Instead of creating a new tracepoint, modify the tracepoint with tracepoint ID number *`n`* by replacing the address expression and the qualifiers of the existing tracepoint to those specified in this command.

Note

You cannot use this qualifier with the `hw_and` qualifier to change a non-chained tracepoint to a chained tracepoint. However, you can modify a chained tracepoint with any other qualifier and also change the address expression.

Alias

`TRCIFETCH` is an alias of `TRACEINSTRFETCH`.

See also

The following commands provide similar or related functionality:

- *ANALYZER* on page 2-28
- *DTRACE* on page 2-139
- *ETM_CONFIG* on page 2-153
- *TRACE* on page 2-288
- *TRACEBUFFER* on page 2-291

- *TRACEDATAACCESS* on page 2-301
- *TRACEDATAREAD* on page 2-307
- *TRACEDATAWRITE* on page 2-313
- *TRACEEXTCOND* on page 2-318
- *TRACEINSTREXEC* on page 2-323.

2.3.137 UNLOAD

The `UNLOAD` command unloads a specified file.

Syntax

`UNLOAD [,`*`qualifier`*`] [{`*`filename`*` | `*`file_num`*`}] [=`*`task`*`]`

where:

`qualifier` If specified, *`qualifier`* must be one of the following:

`all`	Unloads all the files in the file list.
`symbols_only`	Unloads the symbols only, not the executable image.
`image_only`	Unloads the executable image only, not the symbols.

`filename | file_num`

Specifies a file to be unloaded.

Use the `DTFILE` command to list details of the file or files that are associated with the current connection (see *DTFILE* on page 2-137). The details include:

- the file number, which is shown at the start of the output by the text `File` *`file_num`*
- the filename and path.

`task` Applicable only to RTOS, this specifies a task to be unloaded. Use this form of the command if you are running multiple tasks and want to unload only one of them.

Description

The `UNLOAD` command unloads a specified file. If you do not specify a file then all files are unloaded. If you specify a file, using either a filename or a file number, then only that file is unloaded. Any unloaded files remain in the file list and can be reloaded.

The effect of unloading the system file is defined by the vehicle. You can unload only symbols or only the image.

Note

If you have specified any arguments for the image, these are lost when you unload the image. If you specified the arguments as part of the `LOAD` command, you must specify the arguments again when you load the image. Alternatively, after loading the image again, use the `ARGUMENTS` command to specify the arguments.

You do not have to unload an image to run it again. Use the `RESTART` command to reset the PC to the entry point, the use the `GO` command to run the image.

Examples

The following examples show how to use `UNLOAD`:

```
unload dhrystone.axf
```

Unload the symbols (and macros, if any) for the dhrystone program from debugger.

See also

The following commands provide similar or related functionality:

- *ADDFILE* on page 2-23
- *DELFILE* on page 2-116
- *DTFILE* on page 2-137
- *LOAD* on page 2-189
- *RELOAD* on page 2-239
- *RESTART* on page 2-245.

2.3.138 UP

The `UP` command moves up stack levels.

Syntax

```
UP [levels]
```

where:

levels	Specifies the number of levels to climb. If you do not supply a parameter, you move up one level.

Description

The `UP` command moves up stack levels

Each time you move up one level you can see the source line to which you return when you complete execution of your current function or subroutine. At each level you can examine the values of variables and registers that are in scope.

If you are already at the top level a message reminds you that you cannot move up any more. When you have moved up one or more levels, you can use the `DOWN` command (see page 2-131) to move down. When you have moved up one or more levels, any `STEPLINE` or `STEPINSTR` command you issue is effective at the lowest level, not at the level currently in view.

See also

The following commands provide similar or related functionality:

- *DOWN* on page 2-131
- *CONTEXT* on page 2-102
- *DTFILE* on page 2-137.

2.3.139 VCLEAR

The `VCLEAR` command clears a window and sets the cursor to home.

Syntax

```
VCLEAR windowid
```

where:

`windowid`

Specifies the window to be cleared. This must be a user-defined *`windowid`*. See *Window and file numbers* on page 1-5 for details.

Description

The `VCLEAR` command clears a window and sets the cursor to home.

Examples

The following example shows how to use `VCLEAR`:

`vclear 50`	Clear window number 50.

See also

The following commands provide similar or related functionality:

- *PRINTF* on page 2-220
- *VCLOSE* on page 2-337
- *VOPEN* on page 2-343
- *VSETC* on page 2-345.

2.3.140 VCLOSE

The `VCLOSE` command removes and closes a window or file.

Syntax

```
VCLOSE {windowid | fileid}
```

where:

`windowid | fileid`

Specifies the window or file to be closed. This must be a user-defined *`windowid`* or *`fileid`*. See *Window and file numbers* on page 1-5 for details.

Description

The `VCLOSE` command removes and closes a window opened with `VOPEN`, or closes a file opened with `FOPEN`.

Examples

The following example shows how to use `VCLOSE`:

`vclose 50`	Close window number 50.

See also

The following commands provide similar or related functionality:

- *FOPEN* on page 2-166
- *PRINTF* on page 2-220
- *VCLEAR* on page 2-336
- *VOPEN* on page 2-343
- *VSETC* on page 2-345.

2.3.141 VERIFYFILE

The `VERIFYFILE` command compares the contents of a specified file with the contents of target memory.

Syntax

<u>`VERIFYF`</u>`ILE [,{OBJ|raw|rawb|rawhw|ascii[,`*`opts`*`]}]` *`filename`* `[=`*`address/offset`*`]`

where:

`OBJ`	The file is an executable file in the standard target format. RVDK can verify files in either the ELF format or an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The proprietary file format for each version of the RVDK is restricted to the permitted device. This is referred to as the ATPE_*Custom*. There are no *`opts`* supported for this file type.
`raw`	Write the file as raw data, one word per word of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`rawb`	Write the file as raw data, one byte per byte of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`rawhw`	Write the file as raw data, one halfword per halfword of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`ascii`	The file is a stream of ASCII digits separated by whitespace. The interpretation of the digits is specified by other qualifiers (see the *`opts`* qualifier). The starting address of the file must be specified in a one line header in one of the following ways:

`[`*`start`*`]`	The start address.
`[`*`start`*`,`*`end`*`]`	The start address, a comma, and the end address.
`[`*`start`*`,+`*`len`*`]`	The start address, a comma, and the length.
`[`*`start`*`,`*`end`*`,`*`size`*`]`	The start address, a comma, the end address, a comma, and a character indicating the size of each value, where `b` is 8 bits, `h` is 16 bits and `l` is 32 bits.

If the size of the items in the file is not specified, the debugger determines the size by examining the number of white-space separated significant digits in the first data value. For example, if the first data value was `0x00A0`, the size is set to 16-bits.

`opts` Optional qualifiers available for use with the `ascii` qualifier:

`byte` The file is a stream of 8-bit values that are written to target memory without extra interpretation.

`half` The file is a stream of 16-bit values.

`long` The file is a stream of 32-bit values.

`gui` You are prompted to enter the file type with a dialog.

Note

This qualifier has no effect in the headless debugger.

`filename` Specifies the name of the file to be read.

You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myimages`, you can specify:

```
verifyfile,raw,byte "$MYPATH\\myimage.axf" 0x8000...0x8100
```

`address/offset`

Specifies the starting address in target memory for the comparison. If the file being read contains this information, you can adjust it by specifying an offset.

Description

The `VERIFYFILE` command compares the contents of a specified file with the contents of target memory.

Data might be stored in a file in a variety of formats. You can specify the format by specifying the file type. The command then converts the data read from the file before performing the comparison.

The types of file and file formats supported depend on the target processor and any loaded DLLs. The type of memory assumed depends on the target processor. For example, ARM processors have byte addressable memory.

Examples

The following example shows how to use `VERIFYFILE`:

```
verifyfile,ascii,byte "c:\images\rom.dat" =0x8000
```

Verify that the ROM image file in `rom.dat` matches target memory starting at location `0x8000`.

See also

The following commands provide similar or related functionality:

- *READFILE* on page 2-235
- *TEST* on page 2-285
- *WRITEFILE* on page 2-351.

2.3.142 VMACRO

The `VMACRO` command attaches the output of a macro to a window or file.

Syntax

`VMACRO {windowid | fileid} [,macro_name(args)]`

where:

windowid | fileid	Specifies the window of file to be associated with the macro. This must be a user-defined *windowid* or *fileid*. See *Window and file numbers* on page 1-5 for details.
macro_name	Specifies the name and call arguments of the macro that is to send its output to the specified window or file. This happens whenever the macro runs, either directly from the CLI or a command script, or by a breakpoint being hit to which the macro is attached.

Description

The `VMACRO` command attaches a specified macro to a specified window or file. The window or file is updated whenever the macro is called. You can use the following to write data to the window or file:

- `FPRINTF` command (see page 2-169)
- `fputc` predefined macro (see page 3-17)
- `fwrite` predefined macro (see page 3-21).

If you do not supply a macro name, the window or file is disassociated from any macro. The `VMACRO` command runs asynchronously.

Examples

The following examples show how to use `VMACRO`:

`vmacro 50,showmyvars()`	Use the macro `showmyvars()` to write formatted variables to window 50.
`vmacro 50`	Unbind all macros from user window 50.

```
fopen 100,'c:\myfiles\messages.txt' vmacro 100,showmyvars() showmyvars() vmacro
100 vclose 100
```

Use the macro `showmyvars()` to write formatted variables to the file `messages.txt`, unbind the macro from the file, and finally close the file.

See also

The following commands provide similar or related functionality:

- *BREAKACCESS* on page 2-43
- *BREAKEXECUTION* on page 2-52
- *BREAKINSTRUCTION* on page 2-61
- *BREAKREAD* on page 2-69
- *BREAKWRITE* on page 2-78
- *DEFINE* on page 2-110
- *FOPEN* on page 2-166
- *FPRINTF* on page 2-169
- *MONITOR* on page 2-204
- *PRINTVALUE* on page 2-229
- *VOPEN* on page 2-343
- *VSETC* on page 2-345.

The following macros provide similar or related functionality:

- *fopen* on page 3-15
- *fputc* on page 3-17
- *fwrite* on page 3-21.

2.3.143 VOPEN

The `VOPEN` command creates a window that you can use with commands that have a `;`*`windowid`* parameter.

Syntax

```
VOPEN windowid [,screen_num,loc_top,loc_left,loc_bottom,loc_right]
```

where:

`windowid`

Specifies a number to identify the new window. This must be the user-defined *`windowid`* of a window. See *Window and file numbers* on page 1-5 for details.

If a window already exists with the specified number the command fails.

Use this value for the *`windowid`* parameter in commands that you want to display their output in this window.

`screen_num` This parameter is maintained for backward compatibility but is no longer used. If you want to specify the position and size of the new window, you must enter a *`screen_num`* value for the command to parse correctly.

`loc_top` Specifies the number of characters the upper edge of the window is positioned from the top of the screen.

`loc_left` Specifies the number of characters the left side of the window is positioned from the left side of the screen.

`loc_bottom` Specifies the number of characters the bottom row of the window is positioned from the top of the screen.

`loc_right` Specifies the number of characters the right side of the window is positioned from the left side of the screen.

Description

The `VOPEN` command creates a window. When you have created a window you can direct the output from various other commands to it. The commands that can have their output redirected are those that have an optional *`windowid`* parameter.

If you supply only the *`windowid`* parameter, a window is opened with default position and size of 10 rows of 33 characters. The size of a character is determined by the currently selected font so the size and placement of the window might appear to vary between machines and between sessions.

After opening a window you can move and resize it as required.

If the error message `Bad size specification for window` is displayed, check that:

- *`loc_top`* is smaller than *`loc_bottom`*
- *`loc_left`* is smaller than *`loc_right`*
- *`loc_bottom`* and *`loc_right`* are smaller than the screen size.

Examples

The following examples show how to use `VOPEN`:

`vopen 50` Open window number 50 at the default size of 10 rows of 33 characters.

`vopen 50,0,5,5,50,40`

Open window number 50 at position (5,5) and 45 rows of 35 characters.

See also

The following commands provide similar or related functionality:

- *FOPEN* on page 2-166
- *FPRINTF* on page 2-169
- *VCLOSE* on page 2-337
- *VMACRO* on page 2-341
- *WINDOW* on page 2-350.

2.3.144 VSETC

The `VSETC` command positions the cursor in the specified window.

Syntax

```
VSETC windowid ,row ,column
```

where:

windowid	Identifies the window that is to have its cursor positioned. This must be the user-defined *windowid* of a window. See *Window and file numbers* on page 1-5 for details.
row	Specifies the row number in the window, counting from 0, the number of the top row.
column	Specifies the column number in the window, counting from 0, the number of the leftmost column.

Description

The `VSETC` command positions the cursor in the specified window. This defines where the next output to be directed to that window appears.

Example

The following example shows how to use `VSETC`:

```
vsetc 50,2,5
fprintf 50,"Status: %d", status
```

Write `Status:` to window 50, starting from the third column of the sixth row.

See also

The following commands provide similar or related functionality:

- *FOPEN* on page 2-166
- *FPRINTF* on page 2-169
- *VCLOSE* on page 2-337.

2.3.145 WAIT

The WAIT command tells the debugger whether to wait for a command to complete before permitting another command to be issued.

Syntax

```
WAIT = [{ON | OFF}]
```

where:

ON	specifies that all following commands are to run synchronously.
OFF	specifies that following commands run according to their default behavior. This is the default.

Description

The WAIT command makes commands run synchronously. If WAIT is not used, commands use their default behavior.

All commands run from a macro run synchronously unless WAIT is set OFF.

Note

This command requires that RealView Debugger is connected to a debug target.

Example

The following example shows how to use WAIT:

```
wait on
fill/b 0x8000..0x9FFF =0
wait off
```

These commands cause the debugger to fill memory synchronously, forcing you to wait until the fill is complete before accepting another command.

See also

The following command provides similar or related functionality:

- *PAUSE* on page 2-217.

2.3.146 WARMSTART

`WARMSTART` is an alias of `RESET` (see page 2-241).

2.3.147 WHERE

The `WHERE` command displays a call stack.

Syntax

`WHERE [number_of_levels]`

where:

number_of_levels

Specifies the number of levels you want to examine. If you do not supply this parameter, all levels are displayed.

Description

The `WHERE` command displays a call stack. This shows you the function that you are in, and the function that called that, and the function that called that, until the debugger cannot continue. A call stack is not a history of every function call in the life of the process.

The call stack requires debug information for every procedure called. If debug information is not available, the call stack stops. The call stack might also stop prematurely because the stack frames read by the debugger do not conform to the expected structure, for example if memory corruption has occurred, or if a scheduler has created new stack frames.

Examples

The following example shows how to use `WHERE`:

```
> where
#0: (0x24000148) DHRY_2_1\\Proc_7 Line 79. File='C:\Program
Files\ARM\ADSv1_2\Examples\dhry\dhry_2.c'
#1: (0x24000674) DHRY_1_1\\main Line 164. File='C:\Program
Files\ARM\ADSv1_2\Examples\dhry\dhry_1.c'
```

This shows a request for a full stack trace of the dhrystone program. The program was stopped at line 79 of procedure `Proc_7()`. The call stack tells you that this call of `Proc_7()` was made by code at line 164 of `main()`.

The call stack does not tell you what called `main()`. Normally, there is bootstrap code in `__main()` that calls main, but because this code is not normally compiled with debug symbols included, this procedure is not shown in the call stack.

```
> where 1
#0: (0x240002B8) DHRY_1_1\\Proc_3 Line 355. File='C:\Program
Files\ARM\ADSv1_2\Examples\dhry\dhry_1.c'
```

This shows a request for a single level stack trace of the dhrystone program. The program was stopped at line 355 of procedure `Proc_3()`. Compare this to the output of `CONTEXT` at the same location:

```
At the PC: (0x240002B8): DHRY_1_1\Proc_3 Line 355
```

See also

The following commands provide similar or related functionality:

- *CONTEXT* on page 2-102
- *SCOPE* on page 2-252
- *SETREG* on page 2-259.

2.3.148 WINDOW

The `WINDOW` command displays a list of open user-defined windows and files.

Syntax

`WINDOW [{windowid | fileid | name}]`

where

`windowid | fileid`

The user-defined *`windowid`* or *`fileid`*. See *Window and file numbers* on page 1-5 for details.

`name`	The name of the window or file.

Description

The `WINDOW` command displays a list of the user-defined windows that you have opened with the `VOPEN` command, and a list of the user-defined files that you have opened with the `FOPEN` command.

Example

The following command shows a list of files and user-defined windows that are open:

```
> fopen 98,'c:\myfiles\myfile.txt'
> vopen 99
> window
Num     Type    Name
 98     Files   myfile.txt
 99     User    User99
Available Terminal Window types: File, User
```

See also

The following commands provide similar or related functionality:

- *FOPEN* on page 2-166
- *VCLEAR* on page 2-336
- *VCLOSE* on page 2-337
- *VOPEN* on page 2-343.
- *VSETC* on page 2-345.

2.3.149 WRITEFILE

The `WRITEFILE` command writes the contents of memory to a file, performing a format conversion if necessary.

Syntax

<u>`WRITEF`</u>`ILE ,{OBJ|raw|rawb|rawhw|ascii[,`*`opts`*`]}` *`filename =addressrange`*

where:

`OBJ`	Write the file in the standard executable target format. RVDK uses an ELF proprietary file format called *ARM Toolkit Proprietary ELF* (ATPE). The file format for each version of RVDK is restricted to the proprietary ATPE format for the permitted device. This is referred to as ATPE_*Custom*. There are no *`opts`* supported for this file type.
`raw`	Write the file as raw data, one word per word of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`rawb`	Write the file as raw data, one byte per byte of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`rawhw`	Write the file as raw data, one halfword per halfword of memory. There are no *`opts`* supported for this file type. You must specify an address with this qualifier.
`ascii`	Write the file as a stream of ASCII digits separated by whitespace. The exact format is specified by other qualifiers (see the *`opts`* qualifier). The file has a one line header that is compatible with `READFILE` and `VERIFYFILE`. This header has the following format: `[`*`start`*`,`*`end`*`,`*`size`*`]` *`start`* and *`end`* specifies the address range that is written. *`size`* is a character that indicates the size of each value, where `b` is 8 bits, `h` is 16 bits and `l` is 32 bits.
`opts`	Optional qualifiers available for use with the `ascii` qualifier:

`byte`	The file is a stream of 8-bit hexadecimal values that are written to the file without extra interpretation.
`half`	The file is a stream of 16-bit hexadecimal values.

`long`	The file is a stream of 32-bit hexadecimal values.
`gui`	You are prompted to enter the file type with a dialog.

——— **Note** ———

This qualifier has no effect in the headless debugger.

————————

`filename` The name of the file to be written.

You can include one of more environment variables in the filename. For example, if `MYPATH` defines the location `C:\Myfiles`, you can specify:

```
writefile,raw "$MYPATH\\myfile.dat" 0x8000..0x8100
```

`addressrange` The address range in target memory to write to the file. Specify an address range as: *`start_addr..end_addr`*, for example `0x8000..0x9000`.

Description

The `WRITEFILE` command writes the contents of memory to a file, performing a format conversion if necessary.

The type of memory assumed depends on the target processor. For example, ARM processors have byte addressable memory.

Examples

The following examples show how to use `WRITEFILE`:

```
writefile ,raw "c:\temp\file.dat" =0x8000..0x9000
```

Write the contents of the 4KB memory page at `0x8000` to the file `c:\temp\file.dat`, storing the data in raw, uninterpreted, form.

```
writefile ,ascii,long "c:\temp\file.txt" =0x8000..0x9000
```

Write the contents of the 4KB memory page at `0x8000` to the file `c:\temp\file.dat`, storing it as 32-bit values in target memory endianess. For example, the file might look similar to this:

```
[0x8000,0x9000,l]
E28F8090 E898000F E0800008 E0811008
E0822008 E0833008 E240B001 E242C001
E1500001 0A00000E E8B00070 E1540005
...
```

——— **Note** ———

By writing a file as `longs` and reading it back as `longs` on a different target, you can convert the endianness of the data in the file.

————————

See also

The following commands provide similar or related functionality:

- *FILL* on page 2-161
- *LOAD* on page 2-189
- *SETMEM* on page 2-257
- *READFILE* on page 2-235
- *VERIFYFILE* on page 2-338.

Chapter 3
RealView Debugger Predefined Macros

This chapter describes available RealView® Debugger predefined macros. It contains the following sections:

3.1 Predefined macros listed by function

This section lists the commands according to their general function:

- *Access data values at an address*
- *Flow control statements*
- *File and window access* on page 3-3
- *String manipulation* on page 3-3
- *Memory manipulation* on page 3-4
- *Miscellaneous* on page 3-4
- *User interaction macros* on page 3-5.

3.1.1 Access data values at an address

Table 3-1 contains a summary of the predefined macros that return a data value at a given address.

Table 3-1 Access data value macros

Description	Macro
Returns a byte value from the specified address.	`byte` on page 3-7
Returns a long value from the specified address.	`dword` on page 3-9
Returns a word value at the specified address.	`word` on page 3-60

3.1.2 Flow control statements

Table 3-2 contains a summary of the conditional statement macros.

Table 3-2 Flow control statements

Description	Macro
Breaks when an expression evaluates to True.	`until` on page 3-57
Breaks when an expression evaluates to True.	`when` on page 3-59

3.1.3 File and window access

Table 3-3 contains a summary of the file and window access macros.

Table 3-3 File and window access macros

Description	Macro
Returns a byte from file or window.	`fgetc` on page 3-12
Opens a file for reading, writing, or both.	`fopen` on page 3-15
Writes the contents of next byte to a file.	`fputc` on page 3-17
Reads a file into a buffer.	`fread` on page 3-19
Writes a buffer to a file.	`fwrite` on page 3-21

3.1.4 String manipulation

Table 3-4 contains a summary of the string manipulation macros.

Table 3-4 String manipulation

Description	Macro
Concatenates two strings.	`strcat` on page 3-43
Locates the first occurrence of a character in a string.	`strchr` on page 3-45
Compares two strings.	`strcmp` on page 3-47
Copies a string.	`strcpy` on page 3-49
Performs string comparison without case distinction.	`stricmp` on page 3-51
Returns string length.	`strlen` on page 3-53
Performs limited comparison of two strings.	`strncmp` on page 3-55

3.1.5 Memory manipulation

Table 3-5 contains a summary of the memory manipulation macros.

Table 3-5 Memory Manipulation macros

Description	Macro
Searches for a character in memory.	`memchr` on page 3-24
Clears memory values.	`memclr` on page 3-26
Copies characters from memory.	`memcpy` on page 3-28
Sets the value of characters in memory.	`memset` on page 3-30

3.1.6 Miscellaneous

Table 3-6 contains a summary of other predefined macros.

Table 3-6 Miscellaneous macros

Description	Macro
Performs a target function call. Use this to make an application function call from the debugger.	`call` on page 3-8
Processes error message returned from macro.	`error` on page 3-10
Returns a local string from an address.	`getsym` on page 3-23
Returns the value of a specified register.	`reg_str` on page 3-42

3.1.7 User interaction macros

RealView Debugger provides several predefined macros that enable you to get user input or prompt the user to take action. User interaction macros can be used in expressions on the command line and can be called from macros that you create yourself.

Note

Be careful when using these macros as part of test scripts. For example, if you attach the `prompt_text` macro to a breakpoint that is triggered frequently in your program, without testing the return value, it is possible that the debugger displays the prompt message repeatedly in an endless loop.

Table 3-7 contains a summary of the predefined user interaction macros.

Table 3-7 User interaction macros

Description	Macro
Displays a file containing message text.	`prompt_file` on page 3-32
Displays a dialog box containing message text and a choice list.	`prompt_list` on page 3-34
Displays a dialog box containing message text and buttons (**Ok** and **Cancel**).	`prompt_text` on page 3-36
Displays a dialog box containing question text and buttons (**Yes** and **No**).	`prompt_yesno()` on page 3-38
Displays a dialog box containing question text and buttons (**Yes**, **No** and **Cancel**).	`prompt_yesno_cancel()` on page 3-40

3.2 Alphabetical predefined macro reference

This section lists in alphabetical order all the commands that you can issue to RealView Debugger using the CLI.

The following sections describe the available predefined macros:

- `byte` on page 3-7
- `call` on page 3-8
- `dword` on page 3-9
- `error` on page 3-10
- `fgetc` on page 3-12
- `fopen` on page 3-15
- `fputc` on page 3-17
- `fread` on page 3-19
- `fwrite` on page 3-21
- `getsym` on page 3-23
- `memchr` on page 3-24
- `memclr` on page 3-26
- `memcpy` on page 3-28
- `memset` on page 3-30
- `prompt_file` on page 3-32
- `prompt_list` on page 3-34
- `prompt_text` on page 3-36
- `prompt_yesno` on page 3-38
- `prompt_yesnocancel` on page 3-40
- `reg_str` on page 3-42
- `strcat` on page 3-43
- `strchr` on page 3-45
- `strcmp` on page 3-47
- `strcpy` on page 3-49
- `stricmp` on page 3-51
- `strlen` on page 3-53
- `strncmp` on page 3-55
- `until` on page 3-57
- `when` on page 3-59
- `word` on page 3-60.

3.2.1 byte

Returns a byte value from the specified address.

Syntax:

```
unsigned char byte (addr)
void *addr;
```

where:

`addr`	The address containing the value to be returned.

Description

This macro returns a value between 0 and 255, corresponding to the memory contents at the location specified by `addr`. The byte macro uses the indirection operator to obtain the value.

Return Value

`unsigned char`	The byte value located at the specified address.

Rules

The argument default type is specified by using the `OPTION` command (see `OPTION` on page 2-211):

```
OPTION radix = [ decimal | hex ]
```

Example

To display the contents of the hexadecimal address `0x8f5ffff2`, enter the following on the command line:

```
PRINTVALUE byte(0x8f5ffff2)
```

See also

The following macros provide similar or related functionality:

- `dword` on page 3-9
- `word` on page 3-60.

3.2.2 call

Calls a function in the image that is loaded in the target on the current connection.

Syntax:

```
call(label [, arg1,...])
char *label;
int arg1;
```

where:

`label`	The name of the function.
`arg1`	Optional. One or more arguments to the function.

Description

Performs a target function call. Use this to make an application function call from the debugger.

Return Value

None

Rules

None

Example

This example shows how to use `call`:

```
define /R void callTarget()
{
  int val1;
  int val2;
  val1 = 27;
  val2 = 15;
  call(addvalues,val1,val2)
}
.
```

3.2.3 dword

Returns an unsigned long value from a specified address.

Syntax:

```
unsigned long dword (addr)
void *addr;
```

where:

`addr`	The address containing the value to be returned.

Description

This macro returns an unsigned long value, contained within four bytes of memory, corresponding to the memory contents at the location specified by `addr`. The `dword` macro uses the indirection operator to obtain the value.

Return Value

`unsigned long`	The four byte value located at the specified address.

Rules

The argument default type is specified by using the `OPTION` command (see `OPTION` on page 2-211):

```
OPTION radix = [ decimal | hex ]
```

Example

To display the contents of the hexadecimal address `0x8f5ffff2`, enter the following on the command line:

```
PRINTVALUE dword(0x8f5ffff2)
```

See also

The following macros provide similar or related functionality:

- `byte` on page 3-7
- `word` on page 3-60.

3.2.4 error

Processes an error message returned from a macro.

Syntax:

```
int error (type, message, value)
int type;
char *message;
long value;
```

where:

`type`	Specifies the error class, indicated by one of the predefined error codes listed in Table 3-8.

Table 3-8 Error classes

Type	Class	Description
1	note	`message` appears as a line with no prefix. In the GUI, the message appears in the Output pane.
2	warning code	`message` appears with the `Warning:` prefix. In the GUI, the message appears in the Output pane and is also highlighted.
3	error code	In the headless debugger `message` appears with the `Error:` prefix. In the GUI, `message` appears in an Error dialog without the prefix.

`message`	Pointer to char. Points to the first character in a character string for the corresponding error message.
`value`	Variable of type long.

Description

This macro processes an error messages returned from a macro. The error macro generates a call to the error processing function (`_error`). It handles messages from both predefined and user-specified macros.

The message and value parameters are formatted in standard `PRINTF` formats (see `PRINTF` on page 2-220).

Return Value

`int`	Indicates the error message that is displayed in the Cmd tab of the Code window.

Rules

See the `PRINTF` command on page 2-220 for value formats.

Example

Do the following:

1. Define the following macro in an include file, and load the include file:

```
DEFINE /R int odd(n)
  int n;
{
  if ((n & 0x1)==1) // check if number is odd, using
                    // a bitwise AND, and checking for
                    // nonzero result
    return (0);     // zero is returned from this branch,
                    // indicating: Yes, number is odd.
  else              // no - number is even, not odd
    error (2, "number specified (%d) is not odd\n", n);
                    // text msg displayed, %d in format
                    // specifier used for int display,
                    // as in printf()
  return (1);       // 1 is returned when exiting this
                    // branch
}
.
```

2. Enter the command:

```
odd(6)
```

The following error message appears:

```
Warning: number specified (6) is not odd
```

Note

If you run this in the RealView Debugger GUI, the message is displayed in the Output pane and is highlighted.

3.2.5 fgetc

Reads a byte from a file.

Syntax

```
int fgetc (fileid)
int fileid;
```

where:

`fileid`	The ID of the file containing the next byte to be read. This must be a user-defined *`fileid`*. See *Window and file numbers* on page 1-5 for details.

Description

This macro returns the contents of the next byte from a file. The `fgetc` macro name is short for [`file getc()`], where `file` indicates that the macro operates on a file, and `getc` is the standard function for getting a character from a user defined file. This is distinct from the `getchar` function, which can only retrieve a character from the standard input, and is typically the keyboard.

`fgetc` returns the contents of the next memory location byte from the specified file. You define the identity of the file with the `fopen` macro (see `fopen` on page 3-15), or the `FOPEN` command (see `FOPEN` on page 2-166). Any file used to read, or get, the contents of the next byte, must be opened in read mode.

Return value

`int`	Returns the contents of the next byte of memory from a user specified file or window. Returns the value -1 if either an end-of-file mark (`EOF`) or an error is encountered.

Rules

The file read from must be opened in read mode, for example:

```
fopen(100,”c:\\myfiles\\data_in.txt”,”r”)
```

Example

This example shows how to use fgetc, together with fopen and fputc:

```
define /R void copyFile()
{
  int retval;
  int ch;
  // Create data file to read
  retval = fopen(100,”c:\\myfiles\\data_in.txt”,”w”);
  if (retval < 0)
    error(2,"Cannot open file for writing!\n",101);
  else {
    retval = fwrite("1234567890\n1234567890\n1234567890", 1, 32, 100);
    $vclose 100$;
    fopen(100,”c:\\myfiles\\data_in.txt”,”r”);       // open for read-only
    if (retval < 0)
      error(2,"Source file not opened!\n",101);
    else
      fopen(200,”c:\\myfiles\data_out.txt”,”w”);    // open for writing
      if (retval < 0)
        error(2,"Destination file not opened!\n",101);
      else
        do {
          ch = fgetc(100);                            // fgetc()
          if (ch < 0)
            $printf "Finished copying the file!"$;
          else
            fputc(ch,200);                            // fputc()
        } while (ch > 0);
    }
    $vclose 100$;
    $vclose 200$;
  }
}
.
```

See also

The following macros provide similar or related functionality:

- fopen on page 3-15
- fputc on page 3-17
- fread on page 3-19
- fwrite on page 3-21.

The following commands provide similar or related functionality:

- FOPEN on page 2-166

- `FPRINTF` on page 2-169
- `VCLOSE` on page 2-337
- `WINDOW` on page 2-350.

3.2.6 fopen

Opens a file for reading, writing, or both.

Syntax

```
int fopen (fileid, file_name, mode)
int fileid;
char *file_name;
char *mode;
```

where:

fileid	An ID number for the file that is opened. This must be a user-defined *fileid*. See *Window and file numbers* on page 1-5 for details.
`file_name`	A string pointer identifying the name of the file you want to open. If you specify a hardcoded filename you must enclose it in double quotes. See *Rules for specifying filenames* for details on how to specify filenames that include a path.
`mode`	Standard C-style file mode.

Description

This macro opens a file for reading, writing, or both.

Return value

`int` One of the following:

`-1`	Failure
fileid	Success, the ID number of the opened file is returned.

Rules for specifying filenames

Follow these rules when specifying a filename:

- Filenames must be in double quotes, for example "`myfiles/file`".
- Filenames containing a backslash must be in double quotes, with each backslash escaped. For example, “`c:\\myfiles\\file`”.

Example

The example on page 3-13 also shows you how to use `fopen` in a macro.

See also

The following macros provide similar or related functionality:

- `fgetc` on page 3-12
- `fputc` on page 3-17
- `fread` on page 3-19
- `fwrite` on page 3-21.

The following commands provide similar or related functionality:

- `FOPEN` on page 2-166
- `FPRINTF` on page 2-169
- `VCLOSE` on page 2-337
- `VMACRO` on page 2-341
- `WINDOW` on page 2-350.

3.2.7 fputc

Writes a byte to a file.

Syntax

```
int fputc (byte,fileid)
int byte;
int fileid;
```

where:

byte	The byte to be written.
fileid	The ID number of the file where the next byte is to be written. This must be a user-defined *fileid*. See *Window and file numbers* on page 1-5 for details.

Description

This macro writes the contents of the next byte to a file. You must define the identity of the file with either the `fopen` macro (see `fopen` on page 3-15) or the `FOPEN` command (see `FOPEN` on page 2-166). You can also specify the Standard I/O window (20) as output.

Return value

`int`	Not used.

Rules

The file written to must be opened in write mode, for example:

```
fopen(100,”c:\\myfiles\\data_out.txt”,”w”).
```

Example

The example on page 3-13 also shows you how to use `fputc` in a macro.

See also

The following macros provide similar or related functionality:

- `fgetc` on page 3-12
- `fopen` on page 3-15
- `fread` on page 3-19
- `fwrite` on page 3-21.

The following commands provide similar or related functionality:

- `FOPEN` on page 2-166
- `FPRINTF` on page 2-169
- `VCLOSE` on page 2-337
- `WINDOW` on page 2-350.

3.2.8 fread

Reads the contents of a file into a buffer.

Syntax

```
unsigned long fread (buffer, count, size, fileid)
void *buffer;
unsigned count;
unsigned size;
int fileid;
```

where:

`buffer`	Specifies the start of the area into which the data is written.
`count`	Specifies the number of elements.
`size`	Specifies the size of each element in bytes.
`fileid`	The ID number of the file containing the data to be read. This must be a user-defined *`fileid`*. See *Window and file numbers* on page 1-5 for details.

Description

This macro reads the contents of a file into a buffer. You must define the identity of the file with either the `fopen` macro (see `fopen` on page 3-15) or the `FOPEN` command (see `FOPEN` on page 2-166). You can also specify the Standard I/O window (20) as input.

Return value

`unsigned long`

The size of the data that is read, and is the same as `size * count`. However, it returns the value -1 if an end-of-file (`EOF`) or error has occurred.

Rules

None

Example

This example shows how to use `fread` in a macro:

1. Create an include file containing the following macro, for example, `c:\myincludes\myfile.inc`:

```
define /R void readFile(nElements)
  int nElements;
{
  char buffer[37];
  int nbytes;
  int recLen;
  recLen = 6;
  if (nElements > recLen)
    error(2,"Enter a number from 1 to %d.\n",recLen);
  else {
    strcpy(buffer,"One  \nTwo  \nThree\nFour \nFive \nSix  \n");
    fopen(100,"c:\\myfiles\\data.txt","w");
    nbytes = fwrite(buffer, nElements, recLen, 100);
    $printf "%d bytes written\n",nbytes$;
    $vclose 100$;
    fopen(100,"c:\\myfiles\\data.txt","r");
    memset(buffer,0,37);
    nbytes = fread(buffer, nElements, recLen, 100);
    $printf "%d bytes read\n",nbytes$;
    $printf "Strings:\n%s",buffer$;
    $vclose 100$;
  }
}
.
```

2. At the command line, include the file you created in step 1:

   ```
   include 'c:\myincludes\myfile.inc'
   ```

3. Run the macro, specifying a value from 1 to 6:

   ```
   readFile(4)
   ```

See also

The following macros provide similar or related functionality:

- `fgetc` on page 3-12
- `fopen` on page 3-15
- `fputc` on page 3-17
- `fwrite` on page 3-21.

The following commands provide similar or related functionality:

- `FOPEN` on page 2-166
- `FPRINTF` on page 2-169
- `VCLOSE` on page 2-337
- `WINDOW` on page 2-350.

3.2.9 fwrite

Writes the contents of a buffer to a file or window.

Syntax

```
unsigned long fwrite (buffer, count, size, outputid)
void *buffer;
unsigned count;
unsigned size;
int outputid;
```

where:

`buffer`	Specifies the start of the area from which the data is read.
`count`	Specifies the number of elements.
`size`	Specifies the size of each element in bytes.
`outputid`	The ID number of a window or file where the data is to be written. This must be a user-defined *`windowid`* or *`fileid`*. See *Window and file numbers* on page 1-5 for details.

Description

This macro writes the contents of a buffer to a file or window. You must define the identity of the file with either the `fopen` macro (see `fopen` on page 3-15) or the `FOPEN` command (see `FOPEN` on page 2-166). You can also specify the Standard I/O window (20) as output.

Return value

`unsigned long`

The size of the data that is written, and is the same as `size * count`.

Rules

If you are writing to a file, it must be opened in write mode, for example: `fopen(100,”c:\\myfiles\\data_out.txt”,”w”)`.

Example

The example on page 3-19 also shows you how to use `fwrite` in a macro.

See Also

The following macros provide similar or related functionality:

- `fgetc` on page 3-12
- `fopen` on page 3-15
- `fputc` on page 3-17
- `fread` on page 3-19.

The following commands provide similar or related functionality:

- `FOPEN` on page 2-166
- `FPRINTF` on page 2-169
- `VCLOSE` on page 2-337
- `WINDOW` on page 2-350.

3.2.10 getsym

Returns a string from a specified address.

Syntax

```
char *getsym (addr)
void *addr;
```

where:

`addr`	The address containing a string value.

Description

This macro returns a local string from a specified address.

Return value

`char *`	A local string.

Rules

None

Example

This example shows how to use `getsym` on the command line:

```
add char x[20]
strcpy(x,getsym(@pc))
pr x
“Start”
```

3.2.11 memchr

Searches for a character in memory.

Syntax

```
char *memchr (str1, byte_value, count)
char *str1;
char byte_value;
int count;
```

where:

str1	A character pointer to the memory location of the first character byte in a string of characters contained in a file.
byte_value	A character variable used to copy the memory contents of the character occupying a specific position in a character string.
count	An integer variable that specifies the number of characters in `str` that are to be searched for the character specified by `byte_value`.

Description

This macro searches for a character in memory. The `memchr` macro locates the first occurrence of the character `byte_value`, that is contained in the first `count` bytes of memory area that begins with the memory location pointed to by the start variable, `str1`.

Return value

char *	Points to the instance of the character being searched for, called `byte_value`, if one is found. If no instance of the character being searched for is found, then a `NULL` pointer is returned.

Rules

For debugger variables only, a -1 value (`0xFFFFFFFF`) is returned when `byte_value` does not occur in the memory searched on by `memchr`.

Example

This example shows how to use `memchr`:

```
define /R void memoryChr()
{
  char buff[37];
  char *posn;
  strcpy(buff,"1234567890abcdefghijklmnopqrstuvwxyz");
  posn = memchr(buff,'d',20);
  $printf "%s\n",posn$;
}
.
```

See Also

The following macros provide similar or related functionality:

- `memclr` on page 3-26
- `memcpy` on page 3-28
- `memset` on page 3-30.

3.2.12 memclr

Clears memory contents in a specified range.

Syntax

```
char *memclr (str1, count)
char *str1;
int count;
```

where:

`str1`	A character pointer to the memory location of the first character byte in a string of characters that is replaced by the `NUL` character.
`count`	A variable of integer type, used to specify the number of consecutive bytes of memory in a character string that are to be replaced by the `NUL` character.

Description

This macro replaces the specified number of characters in `str1` with the `NUL` character '`\0`' starting at the beginning of `str1`. If `count` is less than the length of `str1`, the macro returns a pointer that points to the address of the character following the area that is cleared.

Return value

`char *`	Points to the first character byte after the string of characters overwritten with the `NUL` character. This enables continuation of the writing process with perfect alignment of bytes for file erasure.

Rules

None

Example

This example shows how to use `memclr`:

```
define /R void memoryClr()
{
  char buff[37];
  char *posn;
  strcpy(buff,"1234567890abcdefghijklmnopqrstuvwxyz");
  posn = memclr(buff,20);
  $printf "%s\n",posn$;
}
.
```

See Also

The following macros provide similar or related functionality:

- `memchr` on page 3-24
- `memcpy` on page 3-28
- `memset` on page 3-30.

3.2.13 memcpy

Copies a specified number of characters from a source memory area to a destination memory area.

Syntax

```
char *memcpy (dest, src, count)
char *dest;
char *src;
int count;
```

where:

`dest`	A character pointer that specifies the starting address for the destination memory area, to begin writing characters to.
`src`	A character pointer that specifies the starting address for the source memory area, to begin copying characters from.
`count`	An integer variable specifying the number of characters (bytes) to be copied, from the source location, to the destination location of memory.

Description

Copies `count` characters from the source memory area, pointed to by `src`, and writes this character string to a destination memory area, pointed to by `dest`.

Return value

`char*`	Points to the destination location that is one byte beyond the last byte written to. This enables continuation of the writing process with perfect alignment of bytes for string concatenation of memory blocks.

Rules

None

Example

This example shows how to use `memcpy`:

```
define /R void memoryCpy()
{
  char buff1[37];
  char buff2[37];
  char *posn;
  strcpy(buff1,"1234567890abcdefghijklmnopqrstuvwxyz");
  posn = memcpy(buff2,buff1,20);
  $printf "%s\n",buff2$;
}
.
```

See Also

The following macros provide similar or related functionality:

- `memchr` on page 3-24
- `memclr` on page 3-26
- `memset` on page 3-30.

3.2.14 memset

Fills a specified area of memory with a character.

Syntax

```
char *memset (dest, byte_value, count)
char *dest;
char byte_value;
int count;
```

where:

`dest`	A character pointer to the memory location where the `memset` macro is to write a string of repeating characters.
`byte_value`	A character variable used to specify the number of times that a character is written consecutively, beginning at the `*dest` address.
`count`	An integer variable used to specify the number of times a particular character is to be written consecutively, beginning with the byte whose address is `*dest`.

Description

This macro writes a character in memory multiple times. The `memset` macro writes the character `byte_value` into the contents of the first count bytes in memory, beginning with the byte pointed to by `dest`. For example, write character `'X'` consecutively, one hundred times, beginning at address `0x8f51fff4`.

Return value

`char *`	Points to the destination location that is one byte beyond the last byte written to. This enables continuation of the writing process with perfect alignment of bytes for string concatenation of memory blocks.

Rules

None

Example

This example shows how to use `memset`:

```
define /R void memorySet()
{
  char buff[37];
  char *posn;
  posn = memset(buff,'@',20);
  $printf "%s\n",buff$;
}
.
```

See Also

The following macros provide similar or related functionality:

- `memchr` on page 3-24
- `memclr` on page 3-26
- `memcpy` on page 3-28.

3.2.15 prompt_file

Displays an Open File dialog.

——— **Note** ———

This macro is not available in the headless debugger.

———————————

Syntax

```
int prompt_file(title, buff)
char *title;
char *buff;
```

where:

`title`	The text that appears in the title bar of the Open File dialog.
`buff`	The filename that appears in the File name text box of the dialog. Assign an empty string to leave the File name text box blank. Contains the chosen path and filename of the opened file when the **Open** button is clicked.

Description

This macro displays an Open File dialog.

Assign an empty string to `buff` to leave the File name text box of the dialog blank.

Assign a filename, without a path, to `buff` before executing this macro, the filename appears in the File name text box of the dialog.

When you click **Open**, `buff` contains the chosen path and filename.

Return value

`int` One of the following:

`0`	File opened.
`1`	Cancel.

Rules

None

Example

This example shows how to use `prompt_file` in a macro:

```
define /R void openFile()
{
  char filename[100];
  int retval;
  retval = prompt_file("Open File", filename);
  if (retval == 1)
    $printf "Open file cancelled!\n"$;
  else
    retval = fopen(100,filename,”r”);    if (retval < 0)
      $printf "Could not open file: %s\n", filename$;
    else
      $printf "Opened file: %s\n", filename$;
}
.
```

See Also

The following macros provide similar or related functionality:

- `prompt_list` on page 3-34
- `prompt_text` on page 3-36
- `prompt_yesno` on page 3-38
- `prompt_yesnocancel` on page 3-40.

3.2.16 prompt_list

Displays a dialog containing message text, a list of choices, and **Ok**, **Cancel** and **Help** buttons.

Note

This macro is not available in the headless debugger.

Syntax

```
int prompt_list (message, buff)
char *message;
char *buff;
```

where:

`message`	The message text that appears at the top of the dialog.
`buff`	Initially, the list of choices that appear for selection in the dialog, separated by `\n`.

Description

This macro displays a dialog containing message text and a list of choices.

Return value

`int` One of the following:

`0`	Cancel
n	Index number of the chosen list item. The first item in the list has an index of 1.

Rules

None

Example

This example shows how to use `prompt_list` on the command line:

```
add char buff[15]
strcpy(buff, “one\ntwo\nthree”)
ce prompt_list(“Choose one:”, buff)Result is: 3 0x03
```

See Also

The following macros provide similar or related functionality:

- `prompt_file` on page 3-32
- `prompt_text` on page 3-36
- `prompt_yesno` on page 3-38
- `prompt_yesnocancel` on page 3-40.

3.2.17 prompt_text

Displays a dialog containing message text, and **Ok** and **Cancel** buttons.

Note

This macro is not available in the headless debugger.

Syntax

```
int prompt_text (message, buff)
char *message;
char *buff;
```

where:

`message`	The message text that appears at the top of the dialog.
`buff`	The buffer that is to contain the user response.

Description

This macro displays a dialog containing message text, and **Ok** and **Cancel** buttons. The user response is entered into the buffer (local or target).

Return value

`int` One of the following:

`0`	OK
`1`	Cancel

Rules

None

Example

This example shows how to use `prompt_text` in a macro:

```
define /R int usrPrompt()
{
  char userPromptBuffer[100];
  int retval;
  retval = prompt_text( "Please enter text", userPromptBuffer );
  if (retval == 0) {
    $printf "Pressed OK\n"$;
    $printf "%s\n", userPromptBuffer$;
  } else
    $printf "Pressed Cancel\n"$;
  return retval;
}
.
```

See Also

The following macros provide similar or related functionality:

- `prompt_file` on page 3-32
- `prompt_list` on page 3-34
- `prompt_yesno` on page 3-38
- `prompt_yesnocancel` on page 3-40.

3.2.18 prompt_yesno

Displays a dialog containing question text, and **Yes** and **No** buttons.

Note

This macro is not available in the headless debugger.

Syntax

```
int prompt_yesno (message)
char *message;
```

where:

`message`	The text that you want to appear as a question on the dialog.

Description

This macro displays a dialog containing question text and two buttons (**Yes** and **No**) for the user reply.

Return value

`int` One of the following:

`0`	Yes
`2`	No

Rules

None

Example

This example shows how to use `prompt_yesno` on the command line:

```
ce prompt_yesno("Is everything OK?")Result is: 0 0x00
```

See Also

The following macros provide similar or related functionality:

- `prompt_file` on page 3-32
- `prompt_list` on page 3-34
- `prompt_text` on page 3-36

- `prompt_yesnocancel` on page 3-40.

3.2.19 prompt_yesno_cancel

Displays a dialog containing question text, and **Yes**, **No** and **Cancel** buttons.

Note

This macro is not available in the headless debugger.

Syntax

```
int prompt_yesno_cancel (message)
char *message;
```

where:

`message`	The text that you want to appear as a question on the dialog.

Description

This macro displays a dialog containing question text and buttons (**Yes**, **No** and **Cancel**) for the user reply.

Return value

`int` One of the following:

`0`	Yes
`1`	Cancel
`2`	No

Rules

None

Example

This example shows how to use `prompt_yesno_cancel` on the command line:

```
ce prompt_yesno_cancel("Is everything OK?")Result is: 1 0x01
```

See Also

The following macros provide similar or related functionality:

- `prompt_file` on page 3-32
- `prompt_list` on page 3-34

- `prompt_text` on page 3-36
- `prompt_yesno` on page 3-38.

3.2.20 reg_str

Returns the value of a specified register.

Syntax

```
unsigned long reg_str (name)
char *name;
```

where:

`name`	The register name.

Description

This macro takes a register name from a string and returns the value for the register.

Return value

`unsigned long`	The value for the register.

Rules

You must be connected to a target.

Example

This example shows how to use `reg_str` on the command line:

```
ce reg_str("@CPSR")
Result is: 211  0xD3
```

3.2.21 strcat

Concatenates two strings.

Syntax

```
char *strcat (dest, src)
char *dest;
char *src;
```

where:

`dest`	A character pointer, that specifies the starting address for the destination memory area. Characters from the `src` string are appended to the end of this string, starting where the `NUL` character previously terminated the string.
`src`	A character pointer that specifies the starting address for the source memory area, to begin copying characters from, when appending to the end of the `*dest` string.

Description

This macro appends the `src` string to the end of the `dest` string, and then returns a pointer to the `dest` string. This macro behaves like the `strcat` function in the ANSI C string library.

Return value

`char *`	Points to the first byte in the `dest` string.

Rules

This macro does not check to see whether the second string can fit in the first array, unless it is a debugger array. Failure to allot enough space for the first array causes excess characters to overflow into adjacent memory locations. Consider using the `strlen` macro first to confirm that there is enough length in `dest`, for the original `dest` and `src` together.

Example

This example shows how to use `strcat` on the command line:

```
add char buff[15]
ce strcpy(buff,"12345")
ce strcat(buff,"67890")
printf "%s\n",buff
1234567890
```

See Also

The following macros provide similar or related functionality:

- `strchr` on page 3-45
- `strcmp` on page 3-47
- `strcpy` on page 3-49
- `stricmp` on page 3-51
- `strlen` on page 3-53
- `strncmp` on page 3-55.

3.2.22 strchr

Locates the first occurrence of a character in a string.

Syntax

```
char *strchr (str1, byte_value)
char *str1;
char byte_value;
```

where:

`str1`	This is a character pointer to the memory location of the first character byte in a string of characters.
`byte_value`	This is a variable of character type, used to specify the character that the `strchr` macro must search for. The terminating `NUL` character '`\0`' is part of the string, so it can be searched for.

Description

This macro locates the first occurrence of a character in a string.

Return value

`char *`	Points to the first memory location of the first occurrence of the character `byte_value` byte. If no instance of the character being searched for is found, then a `NULL` pointer is returned.

Rules

For debugger variables only, a -1 value (`0xFFFFFFFF`) is returned, when `byte_value` does not occur in the string searched on by `strchr`.

Example

This example shows how to use `strchr` in a macro:

```
define /R void substr(character)
  char character;
{
  char *pos;
  pos = strchr("This is a string",character);
  $printf "position: %s\n",pos$;
}
.
```

See Also

The following macros provide similar or related functionality:

- `strcat` on page 3-43
- `strcmp` on page 3-47
- `strcpy` on page 3-49
- `stricmp` on page 3-51
- `strlen` on page 3-53
- `strncmp` on page 3-55.

3.2.23 strcmp

Compares two strings.

Syntax

```
unsigned long strcmp (str1, str2)
char *str1;
char *str2;
```

where:

`str1`	Variable of type pointer to char. Specifies the location in memory of the first byte of a character string.
`str2`	Variable of type pointer to char. Specifies the location in memory of the first byte of a character string.

Description

This macro compares two strings based on the internal machine representation of the characters.

For example, ASCII A has the value 41 in hexadecimal notation and ASCII B has the value 42 in hexadecimal notation. Therefore, A is less than B.

This macro behaves like the `strcmp` function in the ANSI C string library.

Return value

`int` One of the following:

`-1`	Indicates that the second argument string value comes after the first argument string value in the machine collating sequences, `str1` < `str2`.
`0`	Indicates that the two strings are identical in content.
`1`	Indicates that the first argument string value comes after the second argument string value in the machine collating sequences, `str2` < `str1`.

Rules

- Strings are assumed to be `NUL` terminated or to fit within the array boundaries.
- Comparisons are always signed, regardless of how the string is declared.

Example

This example shows how to use `strcmp` on the command line:

```
ce strcmp("string1","string2")
Result is: -1  0xFFFFFFFFFFFFFFFF
```

See Also

The following macros provide similar or related functionality:

- `strcat` on page 3-43
- `strchr` on page 3-45
- `strcpy` on page 3-49
- `stricmp` on page 3-51
- `strlen` on page 3-53
- `strncmp` on page 3-55.

3.2.24 strcpy

Copies a source string into a destination string.

Syntax

```
char *strcpy (dest, src)
char *dest;
char *src;
```

where:

`dest`	A character pointer, that specifies the starting address for the destination character string. Characters from the `src` string are copied to the `dest` string location.
`src`	A character pointer that specifies the starting address for the source character string. This string is copied to the `dest` character string address.

Description

This macro writes the `src` string directly to the `dest` string address beginning at the first byte pointed to by `dest`. It writes until encountering a `NUL` character '`\0`', which designates the end of the `src` string. It returns a pointer to the `dest` string.

This macro behaves like the `strcpy` function in the ANSI C string library.

Return value

`char *`	Points to the first byte in the string `dest`.

Rules

If the destination string is a debugger array, the macro checks the size of the array before copying. Otherwise this macro does not check to see whether the second string can fit in the first array. Failure to allocate enough space for the first array causes excess characters to overflow into adjacent memory locations. Consider using the `strlen` macro first to confirm that there is enough length in `dest`, for the `src` string.

Example

This example shows how to use `strcpy` on the command line:

```
add char buff[50]
ce strcpy(buff,”source string”)
printf ”%s\n”,buff
```

See Also

The following macros provide similar or related functionality:

- `strcat` on page 3-43
- `strchr` on page 3-45
- `strcmp` on page 3-47
- `stricmp` on page 3-51
- `strlen` on page 3-53
- `strncmp` on page 3-55.

3.2.25 stricmp

Compares two strings without case distinction.

Syntax

```
int stricmp (str1, str2)
char *str1;
char *str2;
```

where:

`str1`	Variable of type pointer to char. Specifies the location in memory of the first byte of a character string.
`str2`	Variable of type pointer to char. Specifies the location in memory of the first byte of a character string.

Description

This macro performs string comparison without case distinction. The `stricmp` macro compares strings in ASCII sequence, ignoring case.

Return value

`int`

One of the following:

`-1`	Indicates that the second argument string value comes after the first argument string value in the machine collating sequences, independent of case distinction, `str1` < `str2`.
`0`	Indicates that the two strings are identical in content, independent of case distinction.
`1`	Indicates that the first argument string value comes after the second argument string value in the machine collating sequences, independent of case distinction, `str2` < `str1`.

Rules

- Strings are assumed to be `NUL` terminated or to fit within the array boundaries.
- Comparisons are always signed, regardless of how the string is declared.

Example

This example shows how to use `stricmp` on the command line:

```
ce stricmp("abcDEF",”ABCdef”)
Result is: 0  0x00
```

See Also

The following macros provide similar or related functionality:

- `strcat` on page 3-43
- `strchr` on page 3-45
- `strcmp` on page 3-47
- `strcpy` on page 3-49
- `strlen` on page 3-53
- `strncmp` on page 3-55.

3.2.26 strlen

Returns the length or the specified string.

Syntax

```
unsigned long strlen (str1)
char *str1;
```

where:

`str1`	Variable of type pointer to char. Specifies the location in memory of the first byte of a character string.

Description

This macro returns the string length. The `strlen` macro counts the number of characters in a string up to but not including the `NUL` terminating character.

This macro behaves like the `strlen` function in the ANSI C string library.

Return value

`unsigned long`	Return value is equal to the number of characters in the string pointed to by `str1`, not including the terminating `NUL` character.

Rules

- Strings are assumed to be `NUL` terminated.
- If `str1` is not properly terminated by a `NUL` character, the length returned is invalid.

Example

This example shows how to use `strlen` on the command line:

```
ce strlen("1234567890")
Result is: 10  0x0A
```

See Also

The following macros provide similar or related functionality:

- `strcat` on page 3-43
- `strchr` on page 3-45

- `strcmp` on page 3-47
- `strcpy` on page 3-49
- `stricmp` on page 3-51
- `strncmp` on page 3-55.

3.2.27 strncmp

Performs a limited comparison of two strings.

Syntax

```
int strncmp (str1, str2, count)
char *str1;
char *str2;
int count;
```

where:

`str1`	Variable of type pointer to char. Specifies the location in memory of the first byte of a character string.
`str2`	Variable of type pointer to char. Specifies the location in memory of the first byte of a character string.
`count`	Variable of integer type. Specifies the length of characters in each string to compare, unless the `NUL` character is encountered in either string first.

Description

This macro performs limited comparison of two strings. The `strncmp` macro is used to compare strings in ASCII sequence, except that the comparison stops after count characters, or when the first `NUL` character is encountered, whichever comes first.

This macro behaves like the `strncmp` function in the ANSI C string library.

Return value

`int` One of the following:

`-1`	Indicates that the second argument string value comes after the first argument string value in the machine collating sequences, `str1` < `str2`.
`0`	Indicates that the two strings are identical in content.
`1`	Indicates that the first argument string value comes after the second argument string value in the machine collating sequences, `str2` < `str1`.

Rules

- Strings do not have to be `NUL` terminated or fit within the array boundaries because the comparison is limited to the number of stated characters.

- Less than `count` characters are compared if one of the strings is smaller than count characters.
- The comparison is always signed, regardless of how the string is declared.

Example

This example shows how to use `strncmp` in a macro:

```
define /R void checkfile(filename)
  char *filename;
{
  int retval;
  retval = strncmp(filename, "dhrystone", 4);
  if (retval == 0)
    $printf "%s belongs to the Dhrystone project\n",filename$;
  else
    $printf "%s belongs to another project\n",filename$;
}
.
```

See Also

The following macros provide similar or related functionality:

- `strcat` on page 3-43
- `strchr` on page 3-45
- `strcmp` on page 3-47
- `strcpy` on page 3-49
- `stricmp` on page 3-51
- `strlen` on page 3-53.

3.2.28 until

Breaks when a given expression evaluates to True.

Syntax

```
int until (expression)
int expression;
```

where:

expression	An expression that is evaluated to test if the result is nonzero.

Description

This macro causes execution to break when expression is True. The until macro evaluates its argument, expression, to determine if it is True (nonzero) or False (zero). This macro can only be used with the `GO` command (see `GO` on page 2-173) and the `GOSTEP` command (see `GOSTEP` on page 2-175) to:

- halt execution when the expression passed is True
- continue execution when the expression passed is False.

Return value

`int` One of the following:

`0`	Indicates that expression is False (zero).
`1`	Indicates that expression is True (nonzero).

Rules

Any C expression resulting in a value can be used as the argument, expression.

Example

Set temporary breakpoints at line numbers 3 and 17 in the current module, and at the entry point to the function `printf`. When any of these locations are encountered by the executing program, the debugger stops then checks the until conditional statements. If the variable i is equal to 3 or the variable x is less than y, a break occurs. Otherwise, program execution continues.

```
GO #3,#17,printf;until(i==3||x<y)
```

See also

The following macros provide similar or related functionality:

- `when` on page 3-59

3.2.29 when

Breaks when an expression evaluates to True.

Syntax

```
int when (expression)
int expression;
```

where:

expression	An expression that is evaluated to test if the result is nonzero.

Description

This macro causes execution to break when expression is True. The when macro evaluates its argument, expression, to determine if it is True (nonzero) or False (zero). This macro is designed to be used with any breakpoint commands. When used with this command, program execution halts when the stated expression is True and continues when the stated expression is False.

Return value

`int` One of the following:

`0`	Indicates that expression is True (nonzero).
`1`	Indicates that expression is False (zero).

Rules

Any C expression resulting in a value can be used as the argument, expression.

Example

Set a breakpoint at the entry point of the routine `strcpy`. Each time `strcpy` is encountered, a breakpoint occurs, and the macro `when` is executed. The macro causes program execution to stop when its argument, in this case the byte pointed to by `*str`, is zero.

```
BREAKINSTRUCTION strcpy;when(*str == 0)
```

See also

The following macros provide similar or related functionality:

- `until` on page 3-57

3.2.30 word

Returns a word value at the specified address.

Syntax

```
unsigned short int word (addr)
void *addr;
```

where:

`addr`	The address containing the value to be returned.

Description

This macro returns a word value at the specified address. The word macro returns an unsigned short integer value (a two byte word of memory value) for the contents of memory pointed to by the argument `addr`.

Return value

`unsigned short int`

The two byte value located at the specified address.

Rules

The argument default type is specified by using the `OPTION` command (see `OPTION` on page 2-211):

```
OPTION radix = [ decimal | hex ]
```

Example

To display the contents of the hexadecimal address `0x8f5ffff2`, enter the following on the command line:

```
PRINTVALUE word(0x8f5ffff2)
```

See also

The following macros provide similar or related functionality:

- `byte` on page 3-7
- `dword` on page 3-9.

Chapter 4
RealView Debugger Keywords

This chapter describes the available RealView® Debugger keywords. It contains the following sections:

- *Keywords listed by function* on page 4-2
- *Alphabetical keyword reference* on page 4-4.

4.1 Keywords listed by function

This section lists the keywords according to their general function:

- *Conditional statement keywords*
- *Flow control keywords*
- *Miscellaneous keywords* on page 4-3.

4.1.1 Conditional statement keywords

Table 4-1 contains a summary of the conditional statement keywords.

Table 4-1 Conditional statement keywords

Description	Macro
Simplest form of a conditional statement.	`if` on page 4-10
Specify an alternative statement to execute if the `if` statement evaluates to False.	`if-else` on page 4-11

4.1.2 Flow control keywords

Table 4-2 contains a summary of the conditional statement keywords.

Table 4-2 Flow control keywords

Description	Macro
Exits from the current loop.	`break` on page 4-5
Ignores the remaining statements in the current loop and executes the next iteration of the loop.	`continue` on page 4-6
Executes a given statement one or more times until an expression evaluates to False.	`do-while` on page 4-7
Executes a statement a given number of times.	`for` on page 4-8
Evaluates an expression and executes the following statement or statements until the expression evaluates to False.	`while` on page 4-16

4.1.3 Miscellaneous keywords

Table 4-3 contains a summary of other keywords.

Table 4-3 Miscellaneous keywords

Description	Macro
Verifies that a symbol is currently active.	**`isalive`** on page 4-12
Returns the size of a data type.	**`sizeof`** on page 4-15

You can also use these keywords on the command line by prefixing them with the `CEXPRESSION` command (see `CEXPRESSION` on page 2-91). For example:

```
cexpression sizeof(int)
Result is: 4  0x04
```

4.2 Alphabetical keyword reference

This section lists in alphabetical order the keywords that you can use in macros.

The following sections describe the available keywords:

- **`break`** on page 4-5
- **`continue`** on page 4-6
- **`do-while`** on page 4-7
- **`for`** on page 4-8
- **`if`** on page 4-10
- **`if-else`** on page 4-11
- **`isalive`** on page 4-12
- **`return`** on page 4-14
- s**`izeof`** on page 4-15
- **`while`** on page 4-16.

4.2.1 break

Exits the current loop immediately.

Syntax:

```
break;
```

Description

The **break** statement causes the innermost **for**, **do-while**, or **while** loop to be exited immediately.

Return Value

None

Rules

None

Example

See the example on page 4-9 to see how to use **break** in a **for** loop.

4.2.2 continue

Causes the next iteration of a loop to be executed, ignoring any remaining commands in the loop.

Syntax:

```
continue;
```

Description

The **`continue`** statement causes the remainder of the **`for`**, **`do-while`**, or **`while`** loop to be ignored and the next iteration of the loop to execute.

Return Value

None

Rules

None

Example

See the example on page 4-9 to see how to use **`continue`** in a **`for`** loop.

4.2.3 do-while

Executes one or more statements until an expression is False.

Syntax:

```
do {
    statement;                      /* execute this statement */
    [statement;]...                 /* additional statements */
} while (expression);               /* while this expression is True */
```

where:

expression The expression to be evaluated at the end of each iteration of the loop.

Description

The **do-while** statement executes a given statement one or more times until an expression evaluates to False.

If you have more than one statement in the **do-while** loop these must be enclosed in curly braces ({}).

Return Value

None

Rules

None

Example

This example shows how to use **do-while** in a macro:

```
define /R void doloop()
{
  int i;
  i = 1;
  do {
    $printf "Iteration: %d\n", i$;
    i++;
   } while (i < 11);
}
.
```

4.2.4 for

Executes one or more statements a given number of times.

Syntax:

```
for
    (expression_1;             /* evaluate only once */
     expression_2;             /* evaluate before each iteration */
     expression_3)             /* evaluate after each iteration */
{
    statement;                 /* execute this statement */
                               /* while expression_2 is True */
    [statement;]...            /* additional statements */
}
```

Description

The **`for`** statement is useful for executing a statement a given number of times. It evaluates *`expression_1`* and then evaluates *`expression_2`* to see if it is True, that is nonzero, or False, that is zero. If *`expression_2`* evaluates to True, all statements are executed once.

Next *`expression_3`* is evaluated, and *`expression_2`* is evaluated again to see if it is True or False. If *`expression_2`* is True, all statements are executed again and the cycle continues. If *`expression_2`* is False, all statements are bypassed and execution continues at the next statement outside the **`for`** loop.

Where you have more than one statement in the **`for`** loop these must be enclosed by curly braces ({}).

The term *`expression_1`* can be used to initialize a variable to be used in the loop. It is evaluated once, before the first iteration of the loop. The term *`expression_2`* determines whether to execute or terminate the loop and is evaluated before each iteration. If the term *`expression_2`* evaluates to True, that is nonzero, the loop is executed. If *`expression_2`* is False, that is zero, the loop is terminated. The term *`expression_3`* can be used to increment a loop counter, and is evaluated after each iteration.

Return Value

None

Rules

None

Example

This example shows how to use the **for** statement in a macro:

```
define /R void forloop()
{
  int i;
  for (i=0; i<11; i++) {
    if (i > 10) {
      $printf "    Done!\n"$;
      break;
    } else if (i==5) {
      $printf "    Halfway there...\n"$;
      continue;
    }
    $printf "Iteration: %d\n", i$;
  }
}
.
```

4.2.5 if

The simplest form of a macro conditional statement.

Syntax:

```
if (expression)                 /* If this expression is True */
{
    statement;                  /* execute this statement */
    [statement;]...             /* additional statements */
}
```

Description

The **if** statement is the simplest form of a macro conditional statement. It is always followed by an expression enclosed in parentheses. If the expression evaluates to zero, that is False, the statement following the expression is bypassed. If the expression evaluates to a value other than zero, that is True, the statement following the expression is executed. If you have more than one statement in the **if** statement these must be enclosed in curly braces ({}).

Return Value

None

Rules

None

Example

This example shows how to use **if** in a macro:

```
if (n==0)
  strcpy(number,”zero”);
```

4.2.6 if-else

Provides a way to specify an alternative statement to execute if the **if** statement evaluates to False.

Syntax:

```
if (expression)                   /* If expression is True */
{
    statement_1;                  /* execute statement_1 */
    [statement;]...               /* and these additional statements */
}else                              /* If expression is False */
{
    statement_2;                  /* execute statement_2 */
    [statement;]...               /* and these additional statements */
}
```

Description

The **if-else** statement provides a way to specify an alternative statement to execute if the **if** statement evaluates to False. If the expression evaluates to True, that is nonzero, *statement_1* and any following statements are executed, but *statement_2* and any following statements are not executed. If the expression evaluates to False, that is zero, *statement_2* and any following statements are executed, but *statement_1* and any following statements are not executed. If you have more than one statement in the **if** section or in the **else** section these must be enclosed in curly braces ({}).

Return Value

None

Rules

None

Example

This example shows how to use **if-else** in a macro:

```
if (n==0)
  strcpy(number,”zero”);
else
  strcpy(number,”nonzero”);
```

4.2.7 isalive

Tests the status of the specified symbol.

Syntax

```
int isalive (symbol_name)
  symbol_name;
```

where:

`symbol_name`	The variable name used for the symbol that **`isalive`** tests the scope of.

Description

The **`isalive`** keyword tests whether the specified symbol is in scope. It checks the status of the argument `symbol_name`, to see whether that variable is in scope, and whether it can be referenced. The value returned by **`isalive`** specifies whether the variable does not exist, is not in scope, is in scope inside the current active function, or is an external (global) variable, or a static variable on the stack but out of scope.

Return value

`int` One of the following values

`-1`	Symbol does not exist.
`0`	Symbol not currently active. It cannot be referenced because it is out of scope.
`1`	Symbol currently active. It is part of the local procedure, also called an automatic variable.
`2`	Available on the stack. The symbol is not part of local procedure, and is also called an external (active), global (active), or static automatic variable (inactive, but containing stored memory contents).

Rules

The argument of **`isalive`** must be a variable that has no return value. The argument cannot be a function.

Example

You can use the following syntax for the **`isalive`** keyword when checking the status of a variable used in its argument.

```
CEXPRESSION isalive(xxx)
/* Returns -1 if the symbol xxx does not exist. */
ADD var1
CE isalive(var1)
/* Returns 1 since var1 is defined and active. */
```

Note

The keywords `CEXPRESSION` and `CE` are equivalent.

4.2.8 return

Returns a value from a macro.

Syntax:

```
return [(]expression[)];
```

Description

The **return** statement is used to return a value from a macro. The expression is evaluated, and the resulting value is returned to the caller. If a breakpoint macro returns a value of True, that is nonzero, program execution continues. If it returns a value of False, that is zero, program execution is halted. If a macro never returns a value, the `macro_type` must be declared as `void` when it is defined.

Return Value

None

Rules

None

Example

This example shows how to use **return** in a macro:

```
define /R int value(x)
  int x;
{
  if (x > 0)
    return (x);
  else         // error
    return(-1);
}
.
```

4.2.9 sizeof

Returns the data type size in bytes.

Syntax

`int` **`sizeof`**`(`*`type_name`*`)`

where:

`type_name`	The data type or variable for which the data size is to be determined.

Description

The **`sizeof`** keyword returns the data type size in bytes of a given variable or data type.

Return value

`int`	The size of the data type in bytes.

Rules

None

Example

This example shows how to use **`sizeof`** in a macro:

```
define /R void saveData()
{
  char buffer[37];
  int retval;
  strcpy(buffer,"One  \nTwo  \nThree\nFour \nFive \nSix  \n");
  fopen(100,”c:\\myfiles\\data.txt”,”w”);
  retval = fwrite(buffer, 1, sizeof(buffer)-1, 100);
  $printf "%d bytes written\n",retval$;
  $vclose 100$;
}
.
```

4.2.10 while

Evaluates an expression and executes one or more statements until the expression evaluates to False.

Syntax:

```
while (expression)              /* while this expression is True */
{
    statement;                  /* execute this statement */
    [statement;]...             /* and these additional statements */
}
```

where:

expression	The expression to be evaluated at the start of each loop.

Description

The **while** statement evaluates an expression and executes the following statement or statements until the expression evaluates to False.

The **while** statement must be followed by an expression in parentheses. As long as the expression evaluates to True, all following statements are repeatedly executed. When the expression evaluates to False, all statements are bypassed and execution continues at the next statement outside the **while** loop. If you have more than one statement in the loop these must be enclosed in curly braces ({}).

Return Value

None

Rules

None

Example

This example shows how to use **while** in a macro:

```
define /R void whileloop()
{
  int x;
  x = 1;
  while (1) {
    $printf “Iteration: %d\n”, x$;
```

```
     if (x > 10) {
      $printf “Done!\n”$;
      break;
    } else if (x==5) {
      $printf “Halfway there...\n”$;
    }
    x++;
  }
}
.
```

Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

Access-provider connection	A debug target connection item that can connect to one or more target processors. The term is normally used when describing the RealView Debugger Connection Control window.
Address breakpoint	A type of breakpoint. *See also* Breakpoint.
ADS	*See* ARM Developer Suite.
ARM Developer Suite (ADS)	A suite of software development applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of *RISC* processors. ADS is superseded by RealView Developer Suite (RVDS).
ARM instruction	A word that encodes an operation for an ARM processor operating in ARM state. ARM instructions must be word-aligned.

ARM state	A processor that is executing ARM instructions is operating in ARM state. The processor switches to Thumb state (and to recognizing Thumb instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`. *See also* Thumb state.
Asynchronous execution	*Asynchronous execution* of a command means that the debugger accepts new commands as soon as this command has been started, enabling you to continue do other work with the debugger.
ATPE	ARM Toolkit Proprietary ELF. RVDK uses this file format to read, write and verify executable files. *See also* ELF.
Backtracing	*See* Call Stack.
Big-endian	Memory organization where the least significant byte of a word is at the highest address and the most significant byte is at the lowest address in the word. *See also* Little-endian.
Board	RealView Debugger uses the term *board* to refer to a target processor, memory, peripherals, and debugger connection method.
Board file	The *board file* is the top-level configuration file, normally called `rvdebug.brd`, that references one or more other files.
Breakpoint	A user defined point at which execution stops in order that a debugger can examine the state of memory and registers. *See also* Hardware breakpoint and Software breakpoint.
Call Stack	This is a list of procedure or function call instances on the current program stack. It might also include information about call parameters and local variables for each instance.
Conditional breakpoint	A breakpoint that halts execution when a particular condition becomes True. The condition normally references the values of program variables that are in scope at the breakpoint location.
Context menu	*See* Pop-up menu.

Core module	In the context of Integrator, an add-on development board that contains an ARM processor and local memory. Core modules can run stand-alone, or can be stacked onto Integrator motherboards. *See also* Integrator.
Current Program Status Register (CPSR)	*See* Program Status Register.
Debug Agent (DA)	The Debug Agent resides on the target to provide target-side support for *Running System Debug* (RSD). The Debug Agent implementation is RTOS specific, and can be a thread or built into the RTOS. The Debug Agent and RealView Debugger communicate with each other using the *debug communications channel* (DCC). This enables data to be passed between the debugger and the target using the ICE interface, without stopping the program or entering debug state. *See also* Running System Debug.
Debug With Arbitrary Record Format (DWARF)	ARM code generation tools generate debug information in DWARF2 format by default. From RVCT v2.2, you can optionally generate DWARF3 format (Draft Standard 9).
Deprecated	A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features are to be removed in future versions of the product.
Doubleword	A 64-bit unit of information.
DWARF	*See* Debug With Arbitrary Record Format.
ELF	Executable and Linking Format. ARM code generation tools produce objects and executable images in ELF format.
Embedded Trace Macrocell (ETM)	A block of logic, embedded in the hardware, that is connected to the address, data, and status signals of the processor. It broadcasts branch addresses, and data and status information in a compressed protocol through the trace port. It contains the resources used to trigger and filter the trace output.
EmbeddedICE logic	The EmbeddedICE logic is an on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface. *See also* IEEE1149.1.
Emulator	In the context of target connection hardware, an emulator provides an interface to the pins of a real core (emulating the pins to the external world) and enables you to control or manipulate signals on those pins.

Endpoint connection
A debug target processor, normally accessed through an *access-provider connection*.

ETM *See* Embedded Trace Macrocell.

ETV *See* Extended Target Visibility.

Execution vehicle Part of the debug target interface, execution vehicles process requests from the client tools to the target.

Extended Target Visibility (ETV)
Extended Target Visibility enables RealView Debugger to access features of the underlying target, such as chip-level details provided by the hardware manufacturer or SoC designer.

Floating Point Emulator (FPE)
Software that emulates the action of a hardware unit dedicated to performing arithmetic operations on floating-point values.

FPE *See* Floating Point Emulator.

Halfword A 16-bit unit of information.

Halted System Debug (HSD)
Usually used for RTOS aware debugging, *Halted System Debug* (HSD) means that you can only debug a target when it is not running. This means that you must stop your debug target before carrying out any analysis of your system. With the target stopped, the debugger presents RTOS information to you by reading and interpreting target memory.

See also Running System Debug.

Hardware breakpoint
A breakpoint that is implemented using non-intrusive additional hardware. Hardware breakpoints are the only method of halting execution when the location is in *Read Only Memory* (ROM). Using a hardware breakpoint often results in the processor halting completely. This is usually undesirable for a real-time system.

See also Breakpoint and Software breakpoint.

HSD *See* Halted System Debug.

IEEE Std. 1149.1 The IEEE Standard that defines TAP. Commonly (but incorrectly) referred to as JTAG.

See also Test Access Port

Integrator A range of ARM hardware development platforms. *Core modules* are available that contain the processor and local memory.

Joint Test Action Group (JTAG)

An IEEE group focussed on silicon chip testing methods. Many debug and programming tools use a *Joint Test Action Group* (JTAG) interface port to communicate with processors. For more information see the IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).

JTAG *See* Joint Test Action Group.

JTAG interface unit A protocol converter that converts low-level commands from RealView Debugger into JTAG signals to the EmbeddedICE logic and the ETM.

Little-endian Memory organization where the least significant byte of a word is at the lowest address and the most significant byte is at the highest address of the word.

See also Big-endian.

Pop-up menu Also known as *Context menu*. A menu that is displayed temporarily, offering items relevant to your current situation. Obtainable in most RealView Debugger windows or panes by right-clicking with the mouse pointer inside the window. In some windows the pop-up menu can vary according to the line the mouse pointer is on and the tabbed page that is currently selected.

Processor core The part of a microprocessor that reads instructions from memory and executes them, including the instruction fetch unit, arithmetic and logic unit and the register bank. It excludes optional coprocessors, caches, and the memory management unit.

Profiling Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.

Program Status Register (PSR)

Contains information about the current execution context. It is also referred to as the *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR), which records information about an alternate processor mode.

PSR *See* Program Status Register.

RealView Compilation Tools (RVCT)

RealView Compilation Tools is a suite of tools, together with supporting documentation and examples, that enables you to write and build applications for the ARM family of *RISC* processors.

RealView Debugger Trace

Part of the RealView Debugger product that extends the debugging capability with the addition of real-time program and data tracing. It is available from the Code window.

RealView ICE (RVI) A JTAG-based debug solution to debug software running on ARM processors.

RealView ICE Micro Edition (RVI-ME) ARM’s ICE solution for MCU toolkits.

RSD *See* Running System Debug.

RTOS Real Time Operating System.

Running System Debug (RSD)

Used for RTOS aware debugging, *Running System Debug* (RSD) means that you can debug a target when it is running. This means that you do not have to stop your debug target before carrying out any analysis of your system. RSD gives access to the application using a *Debug Agent* (DA) that resides on the target. The scheduling of the Debug Agent is RTOS specific.

See also Debug Agent and Halted System Debug.

RVCT *See* RealView Compilation Tools.

Scan chain A scan chain is made up of serially-connected devices that implement boundary-scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain. Processors might contain several shift registers to enable you to access selected parts of the device.

Scope The range within which it is valid to access such items as a variable or a function.

Script A file specifying a sequence of debugger commands that you can submit to the command-line interface using the `include` command.

Semihosting A mechanism whereby I/O requests made in the application code are communicated to the host system, rather than being executed on the target.

Software breakpoint A *breakpoint* that is implemented by replacing an instruction in memory with one that causes the processor to take exceptional action. Because instruction memory must be altered software breakpoints cannot be used where instructions are stored in read-only memory. Using software breakpoints can enable interrupt processing to continue during the breakpoint, making them more suitable for use in real-time systems.

See also Breakpoint and Hardware breakpoint.

Software Interrupt (SWI)

An instruction that causes the processor to call a programmer-specified subroutine. Used by the ARM standard C library to handle semihosting.

SPSR Saved Program Status Register.

See also Program Status Register.

Stack traceback This a list of procedure or function call instances on the current program stack. It might also include information about call parameters and local variables for each instance.

SWI *See* Software Interrupt.

Synchronous execution

Synchronous execution of a command means that the debugger stops accepting new commands until this command is complete.

Synchronous starting

Setting several processors to a particular program location and state, and starting them together.

Synchronous stopping

Stopping several processors in such a way that they stop executing at the same instant.

TAP *See* Test Access Port.

TAP Controller Logic on a device which enables access to some or all of that device for test purposes. The circuit functionality is defined in IEEE1149.1.

See also Test Access Port and IEEE1149.1.

Target The target hardware, including processor, memory, and peripherals, real or simulated, on which the target application is running.

Target vehicle Target vehicles provide RealView Debugger with a standard interface to disparate targets so that the debugger can connect easily to new target types without having to make changes to the debugger core software.

Target Vehicle Server (TVS)

Essentially the debugger itself, this contains the basic debugging functionality. TVS contains the run control, base multitasking support, much of the command handling, target knowledge, such as memory mapping, lists, rule processing, board-files and `.bcd` files, and data structures to track the target environment.

Test Access Port (TAP)

The port used to access the TAP Controller for a given device. Comprises **TCK**, **TMS**, **TDI**, **TDO**, and **nTRST** (optional).

Thumb instruction One halfword or two halfwords that encode an operation for an ARM processor operating in Thumb state. Thumb instructions must be halfword-aligned.

Thumb state A processor that is executing Thumb instructions is operating in Thumb state. The processor switches to ARM state (and to recognizing ARM instructions) when directed to do so by a state-changing instruction such as `BX`, `BLX`.

See also ARM state.

TPA *See* Trace port analyzer.

Trace capture hardware

An external device that stores the information from the trace port. Some processors contain their own on-chip trace buffer, where an external device is not required.

Trace port analyzer (TPA)
An external device that stores the information from the trace port. This information is compressed so that the analyzer does not have to capture data at the same bandwidth as that of an analyzer monitoring the core buses directly.

Tracepoint
A tracepoint can be a line of source code, a line of assembly code, or a memory address. In RealView Debugger, you can set a variety of tracepoints to determine exactly what program information is traced.

Tracing
The real-time recording of processor activity (including instructions and data accesses) that occurs during program execution. Trace information can be stored either in a trace buffer of a processor, or in an external trace hardware unit. Captured trace information is returned to the Analysis window in RealView Debugger where it can be analyzed to help identify a defect in program code.

Trigger
In the context of breakpoints, a trigger is the action of noticing that the breakpoint has been reached by the target and that any associated conditions are met.

In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the processor. The exact information that is displayed depends on the position of the trigger within the buffer.

TVS
See Target Vehicle Server.

Vector Floating Point (VFP)
A standard for floating-point coprocessors where several data values can be processed by a single instruction.

VFP
See Vector Floating Point.

Watch
A watch is a variable or expression that you require the debugger to display at every step or breakpoint so that you can see how its value changes. The Watch pane is part of the RealView Debugger Code window that displays the watches you have defined.

Watchpoint
In RealView Debugger, this is a hardware breakpoint.

Word
A 32-bit unit of information.

Index