RealView Debugger
Command Line Reference Guide

Copyright © 2002, 2003 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 2002</td>
<td>A</td>
<td>RealView Debugger 1.5 release.</td>
</tr>
<tr>
<td>September 2002</td>
<td>B</td>
<td>RealView Debugger v1.6 release.</td>
</tr>
<tr>
<td>February 2003</td>
<td>C</td>
<td>RealView Debugger v1.6.1 release.</td>
</tr>
<tr>
<td>September 2003</td>
<td>D</td>
<td>RealView Debugger v1.6.1 release for RVDS 2.0.</td>
</tr>
</tbody>
</table>

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

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

Web Address

http://www.arm.com
Contents

RealView Debugger Command Line Reference Guide

Preface

About this book ........................................................................................................ vi
Feedback ................................................................................................................ x

Chapter 1 Working with the CLI

1.1 About the CLI .................................................................................................... 1-2
1.2 Using the CLI .................................................................................................... 1-6
1.3 Converting legacy scripts to RealView Debugger format .............................. 1-12
1.4 Types of RealView Debugger expressions .................................................. 1-13
1.5 Constructing expressions .............................................................................. 1-14
1.6 Using variables in the debugger ................................................................. 1-20

Chapter 2 RealView Debugger Commands

2.1 Debugger commands listed by function ....................................................... 2-2
2.2 Alphabetical command reference ................................................................. 2-12

Chapter 3 Comparison of Commands

3.1 RealView Debugger, armsd, and AXD commands .................................... 3-2

Glossary
Preface

This preface introduces the RealView™ Debugger v1.6 Command Line Reference Guide. It contains the following sections:

- About this book on page vi
- Feedback on page x.
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-powered 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 Comparison of Commands
Read this chapter for a comparison of the commands supported by RealView Debugger with the commands supported by armsd and AXD.

Glossary
Refer to 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.
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 Debugger documentation suite. Other books in this suite include:

- **RealView Debugger v1.6 Essentials Guide** (ARM DUI 0181)
- **RealView Debugger v1.6 User Guide** (ARM DUI 0153)
- **RealView Debugger v1.6 Target Configuration Guide** (ARM DUI 0182)

Refer to the following books in the RVCT document suite for more information on the compilation tools component of RVDS 2.0:

- **RealView Compilation Tools Essentials Guide** (ARM DUI 0202)
- **RealView Compilation Tools Compiler and Libraries Guide** (ARM DUI 0205)
- **RealView Compilation Tools Linker and Utilities Guide** (ARM DUI 0206)
- **RealView Compilation Tools Assembler Guide** (ARM DUI 0204)

The following documentation provides general information on the ARM architecture, processors, associated devices, and software interfaces:

- **ARM Reference Peripheral Specification** (ARM DDI 0062)

Refer to the following documentation for information relating to the ARM debug interfaces suitable for use with RealView Debugger:
- ARM Agilent Debug Interface User Guide (ARM DUI 0158)

Refer to the following documentation for information relating to specific ARM Limited processors:
- ARM7DI™ Datasheet (ARM DDI 0027)
- ARM710T™ Datasheet (ARM DDI 0086)
- ARM720T™ Datasheet (ARM DDI 0087)
- ARM740T™ Datasheet (ARM DDI 0008)
- ARM7TDMI™ Technical Reference Manual (ARM DDI 0210)
- ARM7EJ-S™ Technical Reference Manual (ARM DDI 0214)
- ARM9TDMI™ Technical Reference Manual (ARM DDI 0180)
- ARM920T™ Technical Reference Manual (ARM DDI 0151)
- ARM922T™ Technical Reference Manual (ARM DDI 0184)
- ARM9EJ-S™ Technical Reference Manual (ARM DDI 0222)
- ARM926EJ-S™ Technical Reference Manual (ARM DDI 0198)
- ARM940T™ Technical Reference Manual (ARM DDI 0144)
- ARM946E-S™ Technical Reference Manual (ARM DDI 0201)
- ARM966E-S™ Technical Reference Manual (ARM DDI 0213)
- ARM1020E™ Technical Reference Manual (ARM DDI 0177)

Refer to the following documentation for details on the FLEXlm license management system, supplied by GLOBEtrotter Inc., that controls the use of ARM applications:
- ARM FLEXlm License Management Guide v3.0 (ARM DUI 0209).

Make sure that you use version 3.0 of this documentation for details on license management in RealView Debugger v1.6.1 for RVDS 2.0.

Other publications

For a comprehensive introduction to ARM architecture see:

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


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


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

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:

- About the CLI on page 1-2
- Using the CLI on page 1-6
- Converting legacy scripts to RealView Debugger format on page 1-12
- Types of RealView Debugger expressions on page 1-13
- Constructing expressions on page 1-14
- Using variables in the debugger on page 1-20.
1.1 About the CLI

RealView Debugger provides two interfaces from which you can perform all supported operations:

**GUI**
The *Graphical User Interface* (GUI) contains a main Code window, in which the majority of features can be accessed. There are also other dialogs and panes you can access from the Code window. For complete details on using the RealView Debugger GUI, see the chapter on the RealView Debugger desktop in the *RealView Debugger Essentials Guide*.

**CLI**
Enables you to enter commands directly into the debugger without requiring the use of GUI items. This chapter provides an introduction to the components of the commands language that you must read if you want to perform debugger operations from the command line. See Chapter 2 *RealView Debugger Commands* for the full list of commands you can enter.

Expressions for the RealView Debugger are written using C language operators and syntax, as described in *Using expressions and statements* on page 1-11.

This section introduces the CLI and contains the following sections:

- Why use the CLI?
- Accessing the CLI on page 1-3
- Getting help on page 1-4.

### 1.1.1 Why use the CLI?

Although you can perform the complete set of RealView Debugger operations from the GUI, there are specific advantages to using the CLI depending on your requirements, such as:

**Setting up the system state**
You can use scripted CLI commands, or include files, to place a target in a particular state, either whenever you connect to a target or on demand. For example, you might use a script to initialize peripherals, or to zero video memory.

**Running test scripts**
Test scripts are used to check that programs are behaving according to specification. You can use memory reads, writes, and target function calls in the CLI to exercise a module API. You can log semihosting output and check that against known correct output, and you can use the debugger to perform soak tests, using many I/O cycles to check system stability.
Performing test script conversion

The CLI enables you to enter commands that are equivalent to armsd and AXD commands. This is particularly useful when you want to convert test scripts you were running with ADS. For details on converting commands, see Converting legacy scripts to RealView Debugger format on page 1-12.

1.1.2 Accessing the CLI

As shown in Figure 1-1, the command line of RealView Debugger is located at the bottom of the interface, directly above the tabs of the Output pane.

![Figure 1-1 Command-line location](image)

Command results and history pane

Command line

Many operations you perform using the GUI have a command-line equivalent, and they are displayed in the command pane as you interact with the GUI. You can perform the same operation later on by entering the command yourself.
1.1.3 Getting help

When you are running the RealView Debugger, you can display details of commands using either of the following:

- **Help on Commands**
- **DCOMMANDS command**.

**Help on Commands**

If you select **Help on Commands** from the **Help** menu in the Code window, the Help window shown in Figure 1-2 opens. This enables you to see a brief description of each command.

![Figure 1-2 Online help on commands](image)

**DCOMMANDS command**

The **DCOMMANDS** command, and its alias **DHELP** and short form **DCOM**, is described fully in **DCOMMANDS** on page 2-94. This command displays the syntax or full description of a single command, a group of related commands, or all available commands. There are some commands that **DCOMMANDS** does not have any information about. For information about these commands see Chapter 2 **RealView Debugger Commands** or use the online help option **Help on Commands**.
As an example, Figure 1-3 shows the use of the `DCOMMANDS` command to display details of the `CONNECT` command.

![Figure 1-3 Displaying details of a command](image-url)
1.2 Using the CLI

This section describes how to enter CLI commands, both manually on the command line, or as batched commands. It also introduces the command syntax supported by the CLI. It contains the following sections:

- Entering commands interactively
- Entering batched commands on page 1-7
- General command language syntax on page 1-8
- Using expressions and statements on page 1-11.

1.2.1 Entering commands interactively

You can enter commands directly onto the command line, which is located at the bottom of the RealView Debugger Code window (see Figure 1-1 on page 1-3).

--- Note ---
For details on entering batched commands, see Entering batched commands on page 1-7.

---

To enter a command at the command line:

1. Select the Output pane so that the command line is given focus and a blinking cursor appears at the command-line prompt.

2. At the prompt >, enter your command using either of the following methods:

   **Entering a new command**
   Type the command you want, followed by any necessary qualifiers or parameters (see Command qualifiers on page 1-9).

   **Reusing a previously entered command**
   If you have already entered a command earlier in the session that is similar, or identical, to the command you want, you can use the up and down arrow keys to display commands previously entered.

   To edit your command, you can use the left and right arrow keys to navigate through the entry, and the Backspace and Delete keys to delete characters.

--- Note ---
The total length of a command must not exceed 4095 characters.

---

3. When you have entered the desired command, press Enter to execute the command.
1.2.2 Entering batched commands

This section describes the various ways in which you can include, and execute, batch (script) files while working with RealView Debugger. You can also use the CLI to invoke a batch file.

You can use RealView Debugger to execute commands from batch files in the following ways:

- **Using the -inc command-line option**
- **Using the INCLUDE command** on page 1-8
- **Using the Command option from the board file advanced info block** on page 1-8.

**Using the -inc command-line option**

RealView Debugger supports its own command-line options, accessible using the Target text field of the Windows shortcut Properties dialog, or from the Unix or Windows shell. These CLI options include `-inc`, which executes the contents of the named file on startup, and `-b`, which controls whether the GUI is displayed or not.

To use `-inc`, you must:

1. Construct a script file for your session, for example using the JOURNAL command on page 2-163.
2. Exit the debugger.
3. Use a Windows shortcut, or a command shell, to run the debugger in batch mode and specify the script file:

   ```
   rvdebug -install="installdir" -b -inc="\path\to\file.txt"
   ```

   **Note**

   - Do not use `-b` without `-inc`. If you use only `-inc`, the script file is run with the GUI enabled.
   - You must specify the install location if the debugger is not in the default directory and you have not defined the install directory using the environment variable `RVDEBUG_INSTALL`.

The CLI startup options are explained in the chapter describing getting started in *RealView Debugger v1.6 User Guide*.
Using the INCLUDE command

The INCLUDE command executes commands that are stored in an include file you specify. For a complete description of this command, and its syntax, see INCLUDE on page 2-161.

Using the Command option from the board file advanced info block

You can configure a board file for each target processor you use with RealView Debugger. The Extended Target Visibility (ETV) definitions for each target are held in the Advanced Information block (see the RealView Debugger Target Configuration Guide for more information).

In the advanced information block, you can use the Command setting to run a single RealView Debugger command when a connection is established. One of the commands you can use here is INCLUDE, which enables you to run more commands from a file.

--- Note ---
Specifying an INCLUDE command in the board file settings requires you to amend your .brd file, usually called rvdebug.brd, located in your default home directory. It is strongly recommended that you make a backup of this file before editing your configuration so that you can restore your configuration file.

1.2.3 General command language syntax

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

- General syntax rules
- Command qualifiers on page 1-9
- Command parameters on page 1-9
- Abbreviations on page 1-10.

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.
- The number of qualifiers and parameters supplied must match the number required or allowed for the command, and these numbers vary between commands.
- You can use any combination of uppercase and lowercase letters in commands.
A command line can be up to 4095 characters in length.

**Command qualifiers**

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

\[ \text{COMMAND} \ [,\text{qualifier} \mid /\text{flag}] \ [\text{parameter}]... \]

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

You introduce each command qualifier with a punctuation character that indicates its type, as follows:

- **,qualifier** A comma introduces a qualifier that provides the debugger with additional information on how to execute a command. A qualifier introduced by a comma is typically a word. For example, the command:
  
  \[ \text{DHELP, FULL =command_name}\]
  
  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*.

- **/flag** A forward slash introduces a flag in the form of 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 \text{FILL} on page 2-144.

  Other flags formed by a slash and one or two letters are included in the command descriptions described in Chapter 2 *RealView Debugger Commands*.

**Command parameters**

As described in **Command qualifiers**, commands accept flags, qualifiers, and parameters.
When entering more than one parameter, you can type a space before each successive parameter to improve readability. If a parameter, for example a filename, includes spaces or other special characters, you must enclose it in double quotes ("..."), or single quotes ('...'). For details on these and all other commands supported by the CLI, see Chapter 2 *RealView Debugger Commands*.

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

- **;window**  
  A semicolon introduces a specification of where any output produced by the command is to be sent. If you supply a ;window parameter, it must be the final parameter of the command.

- **;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. (No command accepts both a ;window and a ;macro-call parameter.)

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 =).

- In high-level mode, code addresses must be referenced by line numbers, labels, and function names, or casted values.

**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 2-23). 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*.

### 1.2.4 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 batch files (see Entering batched commands on page 1-7).

There are many types of expressions accepted by the RealView Debugger CLI, enabling you to further define a command operation 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 1-13.

The RealView Debugger keywords are conditional statements that can be used in a macro definition. For details on creating macros and using them with RealView Debugger, see the chapter on working with macros in the *RealView Debugger User Guide*.

These keywords are the same as the C language program flow keywords, and they cannot be redefined or used in any other context:

- **BREAK**
- **CONTINUE**
- **DO**
- **ELSE**
- **FOR**
- **IF**
- **RETURN**
- **WHILE**.
1.3 Converting legacy scripts to RealView Debugger format

This section describes how you can convert legacy AXD or armsd scripts to the RealView Debugger format. The RealView Debugger product includes two programs to help with this process:

`axd2rvd` Converts an AXD script to RealView Debugger format.

`armsd2rvd` Converts an armsd script to RealView Debugger format.

The commands have the syntax:

```
axd2rvd [-Vsn] [-A] -I infile.asd [-O rvdfile.txt]
armsd2rvd [-Vsn] [-A] -I infile.asd [-O rvdfile.txt]
```

where:

- `-I infile.asd` The name of the existing script file. If you do not specify the input file, the command prints out a usage message. There must be a space between the file option and the filename.

- `-O rvdfile.txt` The name of the new RealView Debugger script file. If you do not specify this, the converted script is written to the standard output, normally the terminal. There must be a space between the file option and the filename.

- `-A` Include input command lines as comments in the output file.

- `-Vsn` Print the version number on the standard output and exit.

You must quote filenames if they contain space characters. The filename extension used for the input or the output files is not predetermined by these programs.

--- Note ---

These commands help you convert a script file to a RealView Debugger include file, but because of the different facilities provided by these debuggers, and the very different command sets implemented, you must check the resulting file for commands that the script cannot convert.

In particular, armsd profiling commands are not converted because the RealView Debugger equivalents are too different.

---

See Chapter 3 Comparison of Commands for a comparison of the commands available in these debuggers.
1.4 Types of RealView Debugger expressions

As described in General command language syntax on page 1-8, 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 1-11.

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>
<thead>
<tr>
<th>Type</th>
<th>Pattern/example</th>
<th>Usage cross-reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetical operation (value or address)</td>
<td>x+(y*5)</td>
<td>FILL on page 2-144</td>
</tr>
<tr>
<td>Array element reference (value or address)</td>
<td>argv[1]</td>
<td>ANALYZER on page 2-26</td>
</tr>
<tr>
<td>Conditional expression</td>
<td>c==3</td>
<td>BREAKINSTRUCTION on page 2-55</td>
</tr>
<tr>
<td>Floating-point expression</td>
<td>3.14</td>
<td>FPRINTF on page 2-151</td>
</tr>
<tr>
<td>Function name reference (code address)</td>
<td>main</td>
<td>LIST on page 2-166</td>
</tr>
<tr>
<td>Line reference (code address)</td>
<td>#line_number</td>
<td>SCOPE on page 2-216</td>
</tr>
<tr>
<td>Macro call</td>
<td>macro()</td>
<td>ALIAS on page 2-23</td>
</tr>
<tr>
<td>Memory address</td>
<td>&amp;address</td>
<td>ADD on page 2-17</td>
</tr>
<tr>
<td>Memory address</td>
<td>(type *) expression</td>
<td>-</td>
</tr>
<tr>
<td>Memory location</td>
<td>0x8000</td>
<td>BREAKREAD on page 2-60</td>
</tr>
<tr>
<td>Memory range expression</td>
<td>0x100..0x200</td>
<td>BREAKREAD on page 2-60</td>
</tr>
<tr>
<td>Qualified line (specifying source module)</td>
<td>MODULE1#12</td>
<td>SCOPE on page 2-216</td>
</tr>
<tr>
<td>Stack level reference</td>
<td>stack_level</td>
<td>SCOPE on page 2-216</td>
</tr>
<tr>
<td>String expression</td>
<td>&quot;string&quot;</td>
<td>FILL on page 2-144</td>
</tr>
<tr>
<td>Symbol reference (value or address)</td>
<td>symbol_name</td>
<td>ADD on page 2-17</td>
</tr>
<tr>
<td>Target connection reference</td>
<td>targetid</td>
<td>CONNECT on page 2-83</td>
</tr>
<tr>
<td>Target program function</td>
<td>function()</td>
<td>BREAKINSTRUCTION on page 2-55</td>
</tr>
</tbody>
</table>


1.5 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 allowed in the debugger (see Using expressions and statements on page 1-11). However, if you are an assembly language user, you do not need 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 1-13. 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 1-15
- Debugger variable symbols on page 1-16
- Macro symbols on page 1-16
- Reserved symbols on page 1-16
- Addresses on page 1-17
- Expression strings on page 1-18
- Target functions on page 1-19.

1.5.1 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 1 024 characters in length. The first character in a symbol must be alphabetic, an underscore _, or the at sign @. The characters allowed 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.
1.5.2 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.

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>
<thead>
<tr>
<th>Form</th>
<th>Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>@root\</td>
<td>@tst\TS1ROOT</td>
<td>References module TS1ROOT in root @tst. (Usually from file loaded as tst.x or tst.out.)</td>
</tr>
<tr>
<td>\global</td>
<td>\x</td>
<td>References global variable x in current root.</td>
</tr>
<tr>
<td>::global</td>
<td>::x</td>
<td></td>
</tr>
<tr>
<td>function\local</td>
<td>main\x</td>
<td>References local variable x in function main.</td>
</tr>
<tr>
<td>MODULE\function</td>
<td>SIEVE\main</td>
<td>References function main in module sieve.</td>
</tr>
<tr>
<td>MODULE\static</td>
<td>SIEVE\y</td>
<td>References static variable y in module sieve.</td>
</tr>
<tr>
<td>MODULE\line_number</td>
<td>ENTRY#18</td>
<td>References line number 18 in module entry.</td>
</tr>
<tr>
<td>MODULE\function\local</td>
<td>ENTRY\main\x</td>
<td>References local variable x in function main in module entry.</td>
</tr>
<tr>
<td>LINE\local</td>
<td>#20\x</td>
<td>References local variable x in an unnamed block at line 20.</td>
</tr>
</tbody>
</table>
1.5.3 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.

1.5.4 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 names of the predefined macros, debugger commands or aliases. See *RealView Debugger User Guide* for more information.

**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 was 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.

1.5.5 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. Since 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 1-17 for a list of reserved symbols and their descriptions.
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 may be all uppercase or all lowercase.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@entry</td>
<td>Used to form a function pseudo-label, function@entry, to ensure that function parameters can be accessed.</td>
</tr>
<tr>
<td></td>
<td>This pseudo-label is defined at the first line of a given function. In general, function@entry refers to</td>
</tr>
<tr>
<td></td>
<td>the first executable line of code in that function or the first auto local that is initialized in that</td>
</tr>
<tr>
<td></td>
<td>function. In either case, function@entry is beyond the function prologue. If no lines exist for the</td>
</tr>
<tr>
<td></td>
<td>function@entry to reference, it is an error. For example, bi function@entry; when (some_arg ==1).</td>
</tr>
<tr>
<td>@hlpc</td>
<td>Indicates your current high-level source code line. @hlpc is valid only if the Program Counter (PC) is in</td>
</tr>
<tr>
<td></td>
<td>a module that has high-level line information (that is, a C, C++, or assembler source module compiled with</td>
</tr>
<tr>
<td></td>
<td>debug turned on).</td>
</tr>
<tr>
<td>@line_range</td>
<td>Contains the line range of the source code associated with the PC.</td>
</tr>
<tr>
<td>@module</td>
<td>Indicates the name of the current module as determined by the location of the PC.</td>
</tr>
<tr>
<td>@function</td>
<td>Indicates the name of the current function as determined by the location of the PC.</td>
</tr>
<tr>
<td>@root</td>
<td>Indicates the name of the current root name.</td>
</tr>
</tbody>
</table>

1.5.6 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 1-4 summarizes the special addressing types supported by the RealView Debugger.

### Table 1-4 Address expressions

<table>
<thead>
<tr>
<th>Addressing type</th>
<th>Indicator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect addresses</td>
<td>[ ]</td>
<td>PRINTVALUE (H W) [23]</td>
</tr>
<tr>
<td>Line numbers</td>
<td>#</td>
<td>BREAKINSTRUCTION #10</td>
</tr>
<tr>
<td>Address ranges</td>
<td>..</td>
<td>DUMP 0X2200..0X2214 DUMP 0X2200..+14</td>
</tr>
<tr>
<td>Multi-statement reference</td>
<td>:</td>
<td>BREAKINSTRUCTION #21:32 (refers to the statement on line 21 that contains column 32)</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>BREAKINSTRUCTION #21.2 (refers to the second statement on line 21)</td>
</tr>
<tr>
<td>Address of non-label symbol</td>
<td>&amp;</td>
<td>BREAKREAD &amp;var</td>
</tr>
</tbody>
</table>

**1.5.7 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 1-5.

### Table 1-5 Examples of expression strings

<table>
<thead>
<tr>
<th>String</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,&quot;abc&quot;</td>
<td>Values 1 and 2, and ASCII values of abc.</td>
</tr>
<tr>
<td>3+4, count, foo()</td>
<td>Value 7, value of count, results of calling foo.</td>
</tr>
<tr>
<td>'1xyz123'</td>
<td>ASCII values of 1, x, y, z, 1, 2, and 3.</td>
</tr>
</tbody>
</table>

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 CE command with that text as its argument. For example:

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

is equivalent to:

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

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

---

### 1.5.8 Target functions

Target functions might 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 as well as 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 will be 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**

```plaintext
CE PROG\strcpy(t,s)
```
1.6 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** on page 1-21
- **Data types** on page 1-21
- **Root names** on page 1-23
- **Module names** on page 1-24
- **Variable references** on page 1-24
- **Stack references** on page 1-26.

1.6.1 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 unless a symbol of the same name exists in the local scope, in which case 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. A symbol local to the current macro (if any).
2. A symbol local to the current line (if any).
3. A symbol local to the current function (if any).
4. A symbol local to the class of the current function (if any).
5. A symbol static to the current module (if any).
6. A global symbol not necessarily in the current module (if any).
7. A static symbol in another module.
8. A global symbol in another root (different loaded file).

1.6.2 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. Each symbol also has an attribute that indicates whether a variable was defined in a code or data area. Further, the assembler can create arrays of these types as well as 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**

```c
(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**

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

This type of casting to an array can be used with the `PRINTVALUE` command. Assembly language structures can be displayed in a more meaningful form by using this technique. Table 1-6 lists additional special casting types. Arrays of hexadecimal types and pointers to hexadecimal types can also be used.

**Table 1-6 Special casting types**

<table>
<thead>
<tr>
<th>Cast</th>
<th>Commands</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(QUOTED STRING)</td>
<td>PRINTVALUE</td>
<td>Show as &quot;string.&quot;</td>
</tr>
<tr>
<td>(Q S)</td>
<td>All</td>
<td>Convert into a legal source-level address.</td>
</tr>
<tr>
<td>(INSTRUCTION ADDRESS)</td>
<td>All</td>
<td>Convert into a single byte.</td>
</tr>
</tbody>
</table>


1.6.3 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 2-17).

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.

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 1-16.

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.

1.6.4 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 ., which 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 modules have the same
name, the debugger appends an underscore followed by a number (that is, PROGA_1,
PROGA_2, PROGA_3, and so on) to the second (and any subsequent) module. To see the
current module and function, use the CONTEXT command.

To print the current module name and/or current function name, use the FPRINTF or
PRINTF command (with the %s format), or use the PRINTVALUE command, with the @module
and @function reserved symbol. You can also use the PRINTSYMBOLS/F command with no
arguments, and the command displays all roots.

The current line number is displayed with %d and the reserved symbol @hlpc. The
reserved symbol hlpc contains the line number at the current PC if located in source
code. Otherwise, it is zero. To print the current source line as text, use %h and @hlpc. To
print the current instruction, use %m and @pc. For more information about debugger
reserved symbols, see Reserved symbols on page 1-16.

1.6.5 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
Examples of variable references are provided in Table 1-7, including an indication of what type of reference is being made.

Table 1-7 Examples of references to variables

<table>
<thead>
<tr>
<th>Variable reference</th>
<th>Reference type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int A;</td>
<td>A is fully referenced.</td>
</tr>
<tr>
<td>A = 5;</td>
<td></td>
</tr>
<tr>
<td>long temp;</td>
<td>temp is fully referenced.</td>
</tr>
<tr>
<td>temp = 9;</td>
<td></td>
</tr>
<tr>
<td>int arr[10], *LABEL;</td>
<td>arr is not fully referenced so its address is used.</td>
</tr>
<tr>
<td>LABEL = arr;</td>
<td>arr[3] is fully referenced.</td>
</tr>
<tr>
<td>arr[3] = 8;</td>
<td></td>
</tr>
<tr>
<td>int AB[10][10], *LABX;</td>
<td>AB is not fully referenced so its address is used.</td>
</tr>
<tr>
<td>LABX = AB[5];</td>
<td>LABEL is fully referenced so its value is used (the address it points to).</td>
</tr>
<tr>
<td>LABX = LABEL;</td>
<td></td>
</tr>
<tr>
<td>char *p, c;</td>
<td>p is fully referenced. c is not fully referenced.</td>
</tr>
<tr>
<td>p = &amp;c;</td>
<td></td>
</tr>
<tr>
<td>c = *LABEL;</td>
<td>LABEL is dereferenced so the value of its address is used.</td>
</tr>
</tbody>
</table>

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 1-8.

Table 1-8 C operators for referencing and dereferencing variables

<table>
<thead>
<tr>
<th>Operator</th>
<th>Scalar</th>
<th>Pointer</th>
<th>Array</th>
<th>Structure</th>
<th>Union</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>-</td>
<td>Ref</td>
<td>Ref</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>Deref</td>
<td>Deref</td>
<td>-</td>
<td>Deref</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-</td>
<td>Ref&lt;sup&gt;a&lt;/sup&gt;</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Ref</td>
<td>Ref</td>
</tr>
<tr>
<td>[ ]</td>
<td>-</td>
<td>Ref</td>
<td>Ref</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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]
```

### 1.6.6 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

With 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 will return, specify the function nesting level preceded by an at sign @. For example, \texttt{GO @1} executes the program until the debugger reaches the address that corresponds to the location where the current function will return to its caller (the instruction after the call). The \texttt{LIST} and \texttt{DISASSEMBLE} commands can be used to show the code at the return address (\texttt{LI @2} for example).

  In nonrecursive programs, the command \texttt{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 \texttt{GO@1; until(depth == 4)} can be used to specify which instance of the address you want (assuming \texttt{depth} is a local variable in your recursive function that determines which 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, \texttt{PRINTVALUE @3\str} references the local variable \texttt{str} of the function at nesting level 3.

- To see all available information about a function, specify the \texttt{EXPAND} command followed by the function nesting level. For example, \texttt{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 will be returned to, and all local variables in the function and their values.
Chapter 2
RealView Debugger Commands

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

- "Debugger commands listed by function" on page 2-2
- "Alphabetical command reference" on page 2-12.
2.1 Debugger commands listed by function

This section lists the commands according to their general function:

- **Board file access**
- **Execution control** on page 2-3
- **Examining source files** on page 2-4
- **Program image management** on page 2-5
- **Target registers and memory** on page 2-6
- **Status enquiries** on page 2-7
- **Macros and aliases** on page 2-7
- **CLI** on page 2-8
- **Program symbol manipulation** on page 2-8
- **Creating and writing to files and windows** on page 2-9
- **Processor tracing** on page 2-10
- **Miscellaneous** on page 2-10.

This section does not include command aliases. See *Alphabetical command reference* on page 2-12 for a full, alphabetical list of commands.

2.1.1 Board file access

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add target definition</td>
<td>ADDBOARD on page 2-20</td>
</tr>
<tr>
<td>Remove target definition</td>
<td>DELBOARD on page 2-100</td>
</tr>
<tr>
<td>Edit current board definition</td>
<td>EDITBOARDFILE on page 2-131</td>
</tr>
<tr>
<td>Read, or reread, a board file</td>
<td>READBOARDFILE on page 2-204</td>
</tr>
<tr>
<td>Select a particular target definition</td>
<td>BOARD on page 2-37</td>
</tr>
<tr>
<td>Connect the debugger to a target</td>
<td>CONNECT on page 2-83</td>
</tr>
<tr>
<td>Disconnect the debugger from a target</td>
<td>DISCONNECT on page 2-111</td>
</tr>
<tr>
<td>List board descriptions</td>
<td>DTBOARD on page 2-118</td>
</tr>
<tr>
<td>Write board memory map as linker file</td>
<td>DUMPMAP on page 2-128</td>
</tr>
</tbody>
</table>
## 2.1.2 Execution control

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
</table>
| Initialize or reset the processor | EMURESET on page 2-132  
RELOAD on page 2-208  
RESET on page 2-210  
RESTART on page 2-213 |
| Configure the DSP simulator | SIMULATOR on page 2-231 |
| Start executing from current state | GO on page 2-154  
RUN on page 2-215 |
| Set a data or instruction breakpoint | BREAKACCESS on page 2-40  
BREAKEEXECUTION on page 2-47  
BREAKINSTRUCTION on page 2-55  
BREAKREAD on page 2-60  
BREAKWRITE on page 2-67 |
| Set a data or instruction tracepoint | TRACE on page 2-250  
TRACEINSTREXEC on page 2-277  
TRACEDATAACCESS on page 2-263  
TRACEDATAREAD on page 2-268  
TRACEDATAWRITE on page 2-273  
TRACEINSTRFETCH on page 2-281 |
| Clear, enable or disable a breakpoint | CLEARBREAK on page 2-79  
DISABLEBREAK on page 2-107  
ENABLEBREAK on page 2-134  
RESETBREAKS on page 2-211 |
| Display currently set breakpoints | DTBREAK on page 2-119 |
| Stop execution at current point | HALT on page 2-158  
STOP on page 2-243 |
| Set or change processor events (for example, exceptions) | BGLOBAL on page 2-33 |
2.1.3 Examining source files

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

Table 2-3 Examining source file commands

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display a specific source file</td>
<td>LIST on page 2-166</td>
</tr>
<tr>
<td>Display execution context</td>
<td>CONTEXT on page 2-89</td>
</tr>
<tr>
<td></td>
<td>UP on page 2-287</td>
</tr>
<tr>
<td></td>
<td>DOWN on page 2-115</td>
</tr>
<tr>
<td></td>
<td>WHERE on page 2-299</td>
</tr>
<tr>
<td>Display locals of an execution context</td>
<td>EXPAND on page 2-140</td>
</tr>
<tr>
<td>Select source or assembly display</td>
<td>MODE on page 2-180</td>
</tr>
<tr>
<td>Move the display location within program</td>
<td>SCOPE on page 2-216</td>
</tr>
<tr>
<td>Display program source files</td>
<td>DTFILE on page 2-121</td>
</tr>
</tbody>
</table>
2.1.4 Program image management

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reload or restart current executable</td>
<td>RELOAD on page 2-208</td>
</tr>
<tr>
<td>Add or remove executable file from loaded files list</td>
<td>ADDFILE on page 2-21</td>
</tr>
<tr>
<td>Load target with one or more executable files</td>
<td>LOAD on page 2-168</td>
</tr>
<tr>
<td>Unload an executable file or process</td>
<td>UNLOAD on page 2-285</td>
</tr>
<tr>
<td>Write to FLASH memory</td>
<td>FLASH on page 2-147</td>
</tr>
<tr>
<td>Translate host filesystem pathnames</td>
<td>NAMETRANSLATE on page 2-183</td>
</tr>
<tr>
<td>Define program (argc, argv)</td>
<td>ARGUMENTS on page 2-30</td>
</tr>
<tr>
<td>Define run mode (RTOS specific)</td>
<td>RUN on page 2-215</td>
</tr>
<tr>
<td>Disassemble target memory</td>
<td>DISASSEMBLE on page 2-109</td>
</tr>
<tr>
<td>Verify data or image file against memory</td>
<td>VERIFYFILE on page 2-290</td>
</tr>
<tr>
<td>Display more information about load errors</td>
<td>DLOADERR on page 2-113</td>
</tr>
<tr>
<td>Do something when execution starts or stops</td>
<td>ONSTATE on page 2-185</td>
</tr>
</tbody>
</table>
2.1.5 Target registers and memory

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable and change target memory layout</td>
<td>MEMMAP on page 2-175</td>
</tr>
<tr>
<td>Fill target memory with value or values</td>
<td>FILL on page 2-144</td>
</tr>
<tr>
<td></td>
<td>SETMEM on page 2-221</td>
</tr>
<tr>
<td></td>
<td>EXPRESSION on page 2-77</td>
</tr>
<tr>
<td>Copy or compare target memory areas</td>
<td>COPY on page 2-91</td>
</tr>
<tr>
<td></td>
<td>COMPARE on page 2-81</td>
</tr>
<tr>
<td>Change target registers</td>
<td>SETREG on page 2-223</td>
</tr>
<tr>
<td>Display memory in window</td>
<td>MEMWINDOW on page 2-178</td>
</tr>
<tr>
<td></td>
<td>LIST on page 2-166</td>
</tr>
<tr>
<td>Disassemble target memory</td>
<td>DISASSEMBLE on page 2-109</td>
</tr>
<tr>
<td>Search for value or values in memory</td>
<td>SEARCH on page 2-218</td>
</tr>
<tr>
<td>Write to FLASH memory</td>
<td>FLASH on page 2-147</td>
</tr>
<tr>
<td>Write memory map to linker file</td>
<td>DUMMAP on page 2-128</td>
</tr>
<tr>
<td>Write host file into target memory</td>
<td>READFILE on page 2-205</td>
</tr>
<tr>
<td>Write target memory to host file</td>
<td>DUMP on page 2-126</td>
</tr>
<tr>
<td></td>
<td>WRITEFILE on page 2-301</td>
</tr>
<tr>
<td>Compare host file with target memory</td>
<td>TEST on page 2-246</td>
</tr>
<tr>
<td></td>
<td>VERIFYFILE on page 2-290</td>
</tr>
</tbody>
</table>

Copyright © 2002, 2003 ARM Limited. All rights reserved.
2.1.6 Status enquiries

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display current process information</td>
<td>DTPROCESS on page 2-123</td>
</tr>
<tr>
<td>Display current image file information</td>
<td>DTFILE on page 2-121</td>
</tr>
<tr>
<td>Display more information about load errors</td>
<td>DLOADERR on page 2-113</td>
</tr>
<tr>
<td>Display execution context</td>
<td>CONTEXT on page 2-89</td>
</tr>
<tr>
<td></td>
<td>WHERE on page 2-299</td>
</tr>
<tr>
<td>Display currently set breakpoints</td>
<td>DTBREAK on page 2-119</td>
</tr>
<tr>
<td>Display trace status</td>
<td>DTRACE on page 2-125</td>
</tr>
<tr>
<td>Display board descriptions</td>
<td>DTBOARD on page 2-118</td>
</tr>
<tr>
<td>Display the contents of a macro</td>
<td>SHOW on page 2-230</td>
</tr>
<tr>
<td>Display simulator instruction pipeline</td>
<td>DPIPEVIEW on page 2-117</td>
</tr>
<tr>
<td>Display and define user preferences</td>
<td>OPTION on page 2-187</td>
</tr>
<tr>
<td></td>
<td>SETTINGS on page 2-226</td>
</tr>
</tbody>
</table>

2.1.7 Macros and aliases

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define a command macro</td>
<td>DEFINE on 2-97</td>
</tr>
<tr>
<td>Invoke a command macro</td>
<td>MACRO on 2-173</td>
</tr>
<tr>
<td>Step invoking a macro at each step</td>
<td>GOSTEP on 2-156</td>
</tr>
</tbody>
</table>
2.1.8 CLI

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

Table 2-8 CLI commands

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run script file</td>
<td>INCLUDE on page 2-161</td>
</tr>
<tr>
<td>Define error action for script file</td>
<td>ERROR on page 2-136</td>
</tr>
<tr>
<td>Cause an abnormal error for script file</td>
<td>FAILINC on page 2-143</td>
</tr>
<tr>
<td>Interrupt current asynchronous command</td>
<td>CANCEL on page 2-76</td>
</tr>
<tr>
<td>Jump (goto) another point in script or macro</td>
<td>JUMP on page 2-165</td>
</tr>
<tr>
<td>Log CLI actions to file</td>
<td>JOURNAL on page 2-163</td>
</tr>
<tr>
<td>Log CLI actions to file</td>
<td>LOG on page 2-171</td>
</tr>
</tbody>
</table>

2.1.9 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

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create symbols referencing target memory</td>
<td>ADD on page 2-17</td>
</tr>
<tr>
<td>Create host-debugger symbols</td>
<td>ADD on page 2-17</td>
</tr>
<tr>
<td>Delete symbols</td>
<td>DELETE on page 2-101</td>
</tr>
<tr>
<td>Browse C++ class structure</td>
<td>BROWSE on page 2-74</td>
</tr>
</tbody>
</table>
### Table 2-9 Program symbol manipulation commands (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load only the symbols for a program</td>
<td>LOAD on page 2-168</td>
</tr>
<tr>
<td>Display symbols in the symbol table</td>
<td>PRINTSYMBOLS on page 2-195</td>
</tr>
<tr>
<td>Display variable type details</td>
<td>PRINTTYPE on page 2-198</td>
</tr>
<tr>
<td>Evaluate expressions involving symbols</td>
<td>CEXPRESSION on page 2-77</td>
</tr>
<tr>
<td></td>
<td>PRINTVALUE on page 2-200</td>
</tr>
<tr>
<td>Display value of symbol every time debugger stops target</td>
<td>MONITOR on page 2-181</td>
</tr>
<tr>
<td></td>
<td>NOMONITOR on page 2-184</td>
</tr>
</tbody>
</table>

### 2.1.10 Creating and writing to files and windows

Table 2-10 shows the commands that manipulate windows.

---

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opening a file</td>
<td>FOPEN on page 2-149</td>
</tr>
<tr>
<td>Creating a new window</td>
<td>VOPEN on page 2-294</td>
</tr>
<tr>
<td>Clearing a window</td>
<td>VCLEAR on page 2-288</td>
</tr>
<tr>
<td>Setting the cursor position within a window</td>
<td>VSETC on page 2-296</td>
</tr>
<tr>
<td>Deleting a window</td>
<td>VCLOSE on page 2-289</td>
</tr>
<tr>
<td>Attaching a macro to a window</td>
<td>VMACRO on page 2-292</td>
</tr>
<tr>
<td>Writing text to a file or window</td>
<td>PRINTF on page 2-192</td>
</tr>
<tr>
<td></td>
<td>FPRINTF on page 2-151</td>
</tr>
<tr>
<td>Commands that support the ;&lt;windowid&gt; parameter</td>
<td></td>
</tr>
</tbody>
</table>
2.1.11 Processor tracing

Table 2-11 shows the processor instruction tracing functions.

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable and disable tracing</td>
<td>TRACE on page 2-250</td>
</tr>
<tr>
<td>Configure the trace capture logic</td>
<td>ANALYZER on page 2-26</td>
</tr>
<tr>
<td></td>
<td>ETM_CONFIG on page 2-137</td>
</tr>
<tr>
<td>Display status information</td>
<td>DTRACE on page 2-125</td>
</tr>
<tr>
<td>Set tracepoints in the program</td>
<td>TRACE on page 2-250</td>
</tr>
<tr>
<td></td>
<td>TRACEDATAACCESS on page 2-263</td>
</tr>
<tr>
<td></td>
<td>TRACEDATAREAD on page 2-268</td>
</tr>
<tr>
<td></td>
<td>TRACEDATAWRITE on page 2-273</td>
</tr>
<tr>
<td></td>
<td>TRACEINSTREXEC on page 2-277</td>
</tr>
<tr>
<td></td>
<td>TRACEINSTRFETCH on page 2-281</td>
</tr>
<tr>
<td>Displaying, saving, and loading captured</td>
<td>TRACEBUFFER on page 2-253</td>
</tr>
<tr>
<td>trace information</td>
<td></td>
</tr>
</tbody>
</table>

2.1.12 Miscellaneous

Table 2-12 shows the remaining functions.

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define user preferences</td>
<td>OPTION on page 2-187</td>
</tr>
<tr>
<td></td>
<td>SETTINGS on page 2-226</td>
</tr>
<tr>
<td>Get help on command</td>
<td>HELP on page 2-159</td>
</tr>
<tr>
<td></td>
<td>DCOMMANDS on page 2-94</td>
</tr>
<tr>
<td>Force debugger to wait for a specified</td>
<td>PAUSE on page 2-191</td>
</tr>
<tr>
<td>number of seconds</td>
<td></td>
</tr>
</tbody>
</table>
### Table 2-12 Miscellaneous commands (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Force debugger to wait, or not to wait, for command to complete</td>
<td>WAIT on page 2-297</td>
</tr>
<tr>
<td>Select thread in RTOS thread group</td>
<td>THREAD on page 2-248</td>
</tr>
<tr>
<td>Quit debugger</td>
<td>QUIT on page 2-203</td>
</tr>
</tbody>
</table>
2.2 Alphabetical command reference

This section lists in alphabetical order all the commands that you can issue to RealView Debugger using the CLI.

Many of the commands have alternative names, or aliases, that you might find easier to remember. This section lists all aliases and includes cross references, where appropriate.

Many 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 ADDBOARD.

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.

The following sections describe the available commands:

- ADD on page 2-17
- ADDBOARD on page 2-20
- ADDFILE on page 2-21
- ALIAS on page 2-23
- ANALYZER on page 2-26
- ARGUMENTS on page 2-30
- BACCESS on page 2-32
- BEXECUTION on page 2-32
- BGLOBAL on page 2-33
- BINSTRUCTION on page 2-36
- BOARD on page 2-37
- BREAD on page 2-39
- BREAK on page 2-39
- BREAKACCESS on page 2-40
- BREAKEXECUTION on page 2-47
RealView Debugger Commands

- BREAKINSTRUCTION on page 2-55
- BREAKREAD on page 2-60
- BREAKWRITE on page 2-67
- BROWSE on page 2-74
- BWRITE on page 2-75
- CANCEL on page 2-76
- CEXPRESSION on page 2-77
- CLEARBREAK on page 2-79
- COMPARE on page 2-81
- CONNECT on page 2-83
- CONTEXT on page 2-89
- COPY on page 2-91
- DBOARD on page 2-93
- DBREAK on page 2-93
- DCOMMANDS on page 2-94
- DEFINE on page 2-97
- DELBOARD on page 2-100
- DELETE on page 2-101
- DELFILE on page 2-103
- DHELP on page 2-106
- DISABLEBREAK on page 2-107
- DISASSEMBLE on page 2-109
- DISCONNECT on page 2-111
- DLOADERR on page 2-113
- DMAP on page 2-114
- DOWN on page 2-115
- DPIPEVIEW on page 2-117
- DTBOARD on page 2-118
- DTBREAK on page 2-119
- DTFILE on page 2-121
- DTPROCESS on page 2-123
- DTRACE on page 2-125
- DUMP on page 2-126
- DUMPMAP on page 2-128
- DVFILE on page 2-130
- DVPROCESS on page 2-130
- EDITBOARDFILE on page 2-131
RealView Debugger Commands

- EMURESET on page 2-132
- EMURST on page 2-133
- ENABLEBREAK on page 2-134
- ERROR on page 2-136
- ETM_CONFIG on page 2-137
- EXPAND on page 2-140
- FAILINC on page 2-143
- FILL on page 2-144
- FLASH on page 2-147
- FOPEN on page 2-149
- FPRINTF on page 2-151
- GO on page 2-154
- GOSTEP on page 2-156
- HALT on page 2-158
- HELP on page 2-159
- HWRESET on page 2-160
- INCLUDE on page 2-161
- JOURNAL on page 2-163
- JUMP on page 2-165
- LIST on page 2-166
- LOAD on page 2-168
- LOG on page 2-171
- MACRO on page 2-173
- MEMMAP on page 2-175
- MEMWINDOW on page 2-178
- MMAP on page 2-179
- MODE on page 2-180
- MONITOR on page 2-181
- NAMETRANSLATE on page 2-183
- NOMONITOR on page 2-184
- ONSTATE on page 2-185
- OPTION on page 2-187
- PATHTRANSLATE on page 2-190
- PAUSE on page 2-191
- PRINTF on page 2-192
- PRINTSYMBOLS on page 2-195
- PRINTTYPE on page 2-198
RealView Debugger Commands

- PRINTVALUE on page 2-200
- PROPERTIES on page 2-202
- PS on page 2-202
- PT on page 2-202
- QUIT on page 2-203
- READBOARDFILE on page 2-204
- READFILE on page 2-205
- REEXEC on page 2-207
- RELOAD on page 2-208
- RESET on page 2-210
- RESETBREAKS on page 2-211
- RESTART on page 2-213
- RSTBREAKS on page 2-214
- RUN on page 2-215
- SCOPE on page 2-216
- SEARCH on page 2-218
- SETFLAGS on page 2-220
- SETMEM on page 2-221
- SETREG on page 2-223
- SETTINGS on page 2-226
- SHOW on page 2-230
- SIMULATOR on page 2-231
- SINSTR on page 2-232
- SM on page 2-232
- SOSTINSTR on page 2-232
- SOVERLINE on page 2-232
- SR on page 2-232
- STEPINDEXTR on page 2-233
- STEPLINE on page 2-235
- STEPOINSTR on page 2-238
- STEPO on page 2-240
- STOP on page 2-243
- SYNCHEXEC on page 2-244
- TEST on page 2-246
- THREAD on page 2-248
- TRACE on page 2-250
- TRACEBUFFER on page 2-253
RealView Debugger Commands

- TRACEDATAACCESS on page 2-263
- TRACEDATAREAD on page 2-268
- TRACEDATAREWRITE on page 2-273
- TRACEINSTREXEC on page 2-277
- TRACEINSTRFETCH on page 2-281
- UNLOAD on page 2-285
- UP on page 2-287
- VCLEAR on page 2-288
- VCLOSE on page 2-289
- VERIFYFILE on page 2-290
- VMACRO on page 2-292
- VOPEN on page 2-294
- VSETC on page 2-296
- WAIT on page 2-297
- WARMSTART on page 2-298
- WHERE on page 2-299
- WRITEFILE on page 2-301
- XTRIGGER on page 2-303.
2.2.1 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 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 2-101 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`:

```plaintext
add mysymbol =3
```

Adds a new symbol called `mysymbol` of type `int` to the debugger symbol table. The new symbol refers to a 1-int area of debugger memory that is given an initial value of 3.

```plaintext
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 2-77
- `DELETE` on page 2-101.
2.2.2 ADDBOARD

The ADDBOARD command creates a board file entry that is not persistent across debugger sessions.

Syntax

```
ADDBOARD name|id [=string,...]
```

where:

- `name` — The name of the board file that you are modifying.
- `string` — Specifies a comma separated list of `name=value` settings to create in the board file entry.

Description

The ADDBOARD command enables you to modify a board file entry in debugger memory, so that you do not have to change the board files themselves.

The command does not modify the file on disk.

See also

The following commands provide similar or related functionality:

- `BOARD` on page 2-37
- `CONNECT` on page 2-83
- `DELBOARD` on page 2-100.
2.2.3 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] name [=\{string,...\}]
```

where:

- **auto** Specifies that only one added file is allowed per process or processor. Any previously added file is removed when the specified file is added.
- **name** The name of the file to be added. You must use double quotes around the name if it contains any spaces.
- **string** The target pathname for 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 REL0AD, described on page 2-208.

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 2-103
- DVFILE on page 2-130
- LOAD on page 2-168
- RELOAD on page 2-208
- UNLOAD on page 2-285.
2.2.4 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. You should therefore 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`:

```plaintext
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 2-47
- `DTFILE` on page 2-121
- `MACRO` on page 2-173.
2.2.5 ANALYZER

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

Syntax

ANALYZER {,config|,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 {,mode_continuous|,mode_trigger|,collect_all|,collect_flow}

where:

config The equivalent of the Select Analysis Configuration option.

edit_properties When not connecting to an ARM ETM-enabled processor, this is
the equivalent of the Configure Analyzer Properties option. To
configure an ETM use ETM_CONFIG.

map_log_phys The equivalent of the Physical to Logical Address Mapping
option. Not available with an ARM ETM-enabled processor.

triggers The equivalent of the Set/Edit Event Triggers option. Not
available with an ARM ETM-enabled processor.

connect The equivalent of the Connect Analysis/Analyzer option. Not
available with an ARM ETM-enabled processor because an ARM
ETM is automatically connected.

set_size:(n) Enable you to set the trace buffer size. 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.

mode_continuous  When no tracepoints are set, capture trace information for everything executed. In other cases, behave as for mode_trigger.

mode_trigger  When tracepoints are set, capture trace information as determined by the tracepoint.

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.

Description

The ANALYZER command and, with the ETM 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.
The options are split into several groups:

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

- 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 `mode_continuous` and `mode_trigger` toggle between requiring a tracepoint to collect trace data and its continuous collection.

- 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:

- `DTRACE` on page 2-125
- \textit{ETM\_CONFIG} on page 2-137.
- \textit{TRACE} on page 2-250
- \textit{TRACEBUFFER} on page 2-253
- \textit{TRACEDATAREAD} on page 2-268
- \textit{TRACEDATAACCESS} on page 2-263
- \textit{TRACEDATAWRITE} on page 2-273
- \textit{TRACEINSTREEXEC} on page 2-277
- \textit{TRACEINSTRFETCH} on page 2-281.
2.2.6 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.

If a literal double-quote character is required in the arguments, it must be quoted using the backslash character.

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
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.afl"
- argv[1] -f
- argv[2] "my file.c"
- argv[3] -o
- argv[4] "my file.o"
- argv[5] NULL

See also

The following commands provide similar or related functionality:

- GO on page 2-154
- LOAD on page 2-168
- RESTART on page 2-213.
2.2.7 BACCESS

BACCESS is an alias of BREAKACCESS (see page 2-40).

2.2.8 BEXECUTION

BEXECUTION is an alias of BREAKEXECUTION (see page 2-47).
2.2.9 BGLOBAL

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

Syntax

BGLOBAL [request] [name] [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 allowing you to select a global breakpoint to enable or disable.

*name* Identifies the global breakpoint to be enabled or disabled.

*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 true (nonzero) value, execution continues. If the macro returns a false (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 events see the processor architecture manual.

Some simulators, including ARMulator, can extend the list of possible breakpoint events beyond that defined for the processor. These extensions are normally defined by peripheral or memory models included in the simulator. For example, a memory model might define a DMA transfer event.

Each extra event is named by the model that implements it, and these names are displayed with the standard names in the GUI. You can set and modify global breakpoints for these events using the BGLOBAL command by specifying the event name as *name* in the command. If the name includes spaces, you must enclose it in double quotes.
Compatibility

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

ARM architecture processors

The possible events are the exception types supported by the processor. The supported names are:

- **Reset**: The RESET exception.
- **Undefined**: The undefined instruction exception.
- **SWI**: The software interrupt (SWI) exception.
- **P_Abort**: The prefetch abort (instruction memory read abort) exception.
- **D_Abort**: The data abort (data memory read or write abort) exception.
- **Address**: The address exception. Used by the now obsolete 26-bit ARM processor architectures.
- **IRQ**: The interrupt request exception.
- **FIQ**: The fast interrupt request exception.
- **Error**: The error exception.

Oak and TeakLite processors

Global breakpoints (processor events) are not supported for DSP simulators. The command returns:

Error 50019 (Server): Breakpoint type is not supported.

The events supported by the processor hardware are:

- **ILL**: The illegal access break.
- **BKR**: The block repeat break.
- **TBF**: The trace buffer full break.
- **INT**: The interrupt break.
- **BRE**: The branch break.
- **EXR**: The external register read break.
- **EXW**: The external register write break.

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 should the application start executing data literals (the usual reason for unintentionally executing an undefined instruction) you can find out why:

bglobal,enable undefined

Some processor events interact with other debugger functions. For example, the ARM SWI exception is used by the ARM Semihosting interface, and the Oak processor trace buffer full break is used by the Oak trace configuration mode.

See also

The following command provides similar or related functionality:

- GO on page 2-154.
2.2.10 BINSTRICTION

BINSTRICTION is an alias of BREAKINSTRUCTION (see page 2-55).
2.2.11 BOARD

The BOARD command changes the current board, also known as the current connection. By default, all actions apply to the current connection.

Note

The BOARD command is only available when RealView Debugger is licensed for multiple processor use.

Syntax

BOARD [,next][,default][=resource]

where:

next   Connects the debugger to the next board listed in the board list.

default Connects the debugger to the board that is listed first in the board file.

resource Identifies the board that is to become the current board. Specify an integer number that identifies the board. The = in this parameter is optional.

Description

With no parameters, the BOARD command displays the name of the current board. With one of the qualifiers or the resource argument, it connects the debugger to either:

• the specified board
• the next board in the board list
• the default board.

The newly selected board becomes the current target connection. RealView Debugger uses the term board because a target connection is defined by more than a processor device. The memory map and the available peripherals are normally defined by the target as a whole, and so it is more appropriate to refer to boards than to processors.

You can display the boards that you can cycle through using board,next by clicking the connection drop-down list from the toolbar.

Using ADDBOARD, you can add a board to the list of boards that can be selected when creating a new connection, for the duration of the current session. You can remove an unconnected board from the list using DELBOARD. To add or remove a board from the board list permanently, you must edit the board file using EDITBOARDFILE.
Examples

The following examples show how to use BOARD:

board, next
Changes the current board to the next defined board from the board list.

board = 2
Changes the current board to be board number 2 in the board list.

board = "ARM966E-S_1:ARM-A-RR"
Changes the current board to the named board in the board list.
The name of the current board is displayed in the titlebar of the Code Window and in the output of board command with no arguments.

See also

The following commands provide similar or related functionality:

- ADDBOARD on page 2-20
- CONNECT on page 2-83
- DELBOARD on page 2-100
- DISCONNECT on page 2-111
- EDITBOARDFILE on page 2-131
- THREAD on page 2-248.
2.2.12 BREAD

BREAD is an alias of BREAKREAD (see page 2-60).

2.2.13 BREAK

BREAK is an alias of BREAKINSTRUCTION (see page 2-55).
2.2.14 BREAKACCESS

The BREAKACCESS command sets an access (memory read or memory write) breakpoint at the specified memory location(s).

Syntax

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

where:

- `qualifier` Is a list of zero or more qualifiers. The possible qualifiers are described in Description.
- `address` or `address_range` Specifies a single address in target memory, or 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 TRUE (nonzero) value, or you specified `continue` in the qualifiers, execution continues. If the macro returns a FALSE (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 2-119, 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 the starting and stopping other processors or threads.
If supported by an RTOS kernel, these breakpoints can be thread-specific, although usually not for hardware-assisted breaks.

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 fails, target execution resumes with the instruction at the breakpointed location.
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. Macros specified with `macro:` are run in this phase.
5. If there is a macro specified after a semicolon on the command line, this is run.
6. If the qualifiers included `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.

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 not allowed for a specific processor or vehicle, but this is determined when you issue the command. 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.

- `continue`
  
  Any triggering of the breakpoint is recorded (for example, as a journal entry, or by the action of a macro) but execution then continues.

- `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:

  ```
  ba,context:{\HELLO_1},when:{status>0} #15
  ```

  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:
ba,when:{\HELLO_1\HELLO_C\status>0} \HELLO_1\HELLO_C\#15

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.

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:0x440 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:

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 0x400-0x4F0:

\[
\text{BREAKACCESS, hw_dvalue:0x440, hw_dmask:0xF0F 0x1FA00}
\]

**hw_passcount:** (n)

Specifies the number of times that the specified condition has to occur to trigger the breakpoint. The default value is 1. 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 supplement the hardware one for higher count values. When both hardware and software pass counts are defined:

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, \( \text{hw_and:}{2} \), or \( \text{hw_and:}{\text{then}-2} \), where 2 is the breakpoint id of another breakpoint.

In the and form, the conditions associated with both breakpoints are tied, 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
- prev for the last breakpoint specified for this connection.

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.
**RealView Debugger Commands**

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 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.

message:{message}

Triggering of the breakpoint results in message being output. Prefixing message with $n$ enables you to write the message text to custom window n, where n is between 50-1024.

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.

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 specified condition has to occur to trigger the breakpoint. The default value is 1. 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:

```
ba, register: PR1
```

or

```
ba, register: @PR1
```

**Note**  
You cannot specify core registers with this qualifier.

**sample**  
Not supported in this release.

**timed**  
Triggering records the time, in whatever units the debug hardware chooses, from the last reset of time. The time can be in nanoseconds, micro-seconds, processor cycles, instructions, or just 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 will keep a log of times for each break.

**update:** `{@}`  
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.

when_not:{condition}

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

**Alias**

BAACCESS is an alias of BREAKACCESS.

**See also**

The following commands provide similar or related functionality:

- BREAKEXECUTION on page 2-47
- BREAKINSTRUCTION on page 2-55
- BREAKREAD on page 2-60
- BREAKWRITE on page 2-67
- CLEARBREAK on page 2-79
- DTBREAK on page 2-119
- ENABLEBREAK on page 2-134.
2.2.15 BREAKEXECUTION

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

Syntax

```
BREAKEXECUTION [{, qualifier...}] expression [= {threads, ...}] []; macro-call
```

where:

- **qualifier** Is an ordered list of zero or more qualifiers. The possible qualifiers are described in Description.
- **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** Not supported in this release.
- **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 TRUE (nonzero) value, or you specified continue in the qualifiers, execution continues. If the macro returns a FALSE (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 2-119, 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.

If supported by an RTOS kernel, these breakpoints can be thread-specific, although usually not for hardware-assisted breaks.

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 fails, target execution resumes with the instruction at the breakpointed location.
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. Macros specified with macro: are run in this phase.
5. If there is a macro specified after a semicolon on the command line, this is run.
6. If the qualifiers included 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.

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 not allowed for a specific processor or vehicle, but this is determined when you issue the command. 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.
- continue Any triggering of the breakpoint is recorded (for example, as a journal entry, or by the action of a macro) but execution then continues.
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:
```
be,context:{\HELLO_1\HELLO_C},when:{status>0} #15
```
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:
```
be,when:{\HELLO_1\HELLO_C\status>0} \HELLO_1\HELLO_C\#15
```

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.

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:
```
BE,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:
```
BE,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:
```
BE,hw_dvalue:0x440 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 \( 0x00 - 0x18 \):

\[
\text{BE, hw_dvalue:0x0, hw_dhigh:0x18 0x1000}
\]

\[
\text{hw_dmask:}(n)
\]

Specifies the data value mask value for a data-range breakpoint. Data values that match the value specified by the \text{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 \( 0x400 - 0x4F0 \):

\[
\text{BE, hw_dvalue:0x440, hw_dmask:0xF0F 0x1FA00}
\]

\[
\text{hw_passcount:}(n)
\]

Specifies the number of times that the specified condition has to occur to trigger the breakpoint. The default value is 1. This qualifier differs from \text{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 supplement the hardware one for higher count values. When both hardware and software pass counts are defined:

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

\[
\text{hw_and:{[then-]}id}
\]

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

In the \text{and} form, the conditions associated with both breakpoints are tied, so that the action associated with the second breakpoint are performed only when both conditions simultaneously match.

In the \text{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 \text{continue} qualifier is specified in the second breakpoint, the program stops.
The *id* is one of:

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

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 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.

```
BE, hw_not: addr 0x10040
```

This means to break at any address other than 0x10040. This 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.

`message: {message}`  
Triggering of the breakpoint results in *message* being output. Prefixing *message* with `$n$` enables you to write the message text to custom window *n*, where *n* is between 50-1024.
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.

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 specified condition has to occur to trigger the breakpoint. The default value is 1. 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.

timed

Triggering records the time, in whatever units the debug hardware chooses, from the last reset of time. The time can be in nanoseconds, micro-seconds, processor cycles, instructions, or just 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 will keep 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.
RealView Debugger Commands

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.

\textbf{when}:\{condition\}

The breakpoint is triggered whenever \textit{condition}, a debugger expression, evaluates to \texttt{TRUE}.

\textbf{when_not}:\{condition\}

The breakpoint is triggered whenever \textit{condition}, a debugger expression, evaluates to \texttt{FALSE}.

\section*{Examples}

The following examples show how to use \texttt{BREAKEXECUTION}:

\begin{verbatim}
BREAKEXECUTION \MATH_1\#449.3
Set a hardware breakpoint at line 449, statement 3 in the file math.c.

BREAK \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 \texttt{TRUE}, 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 \texttt{FALSE}, the debugger continues application execution.
\end{verbatim}
Alias

BEEXECUTION is an alias of BREAKEXECUTION.

See also

The following commands provide similar or related functionality:

- BREAKINSTRUCTION on page 2-55
- MACRO on page 2-173.
2.2.16 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.

Syntax

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

where:

- qualifier Is an ordered list of zero or more qualifiers. The possible qualifiers are described in Description
- 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 Not supported in this release.
- 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 TRUE (nonzero) value, or you specified continue in the qualifiers, execution continues. If the macro returns a FALSE (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 2-119, 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.

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 fails, target execution resumes with the instruction at the breakpointed location.
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. Macros specified with macro: are run in this phase.
5. If there is a macro specified after a semicolon on the command line, this is run.
6. If the qualifiers included 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.

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 not allowed for a specific processor or vehicle, but this is determined when you issue the command. 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.

- **continue**
  - Any triggering of the breakpoint is recorded (for example, as a journal entry, or by the action of a macro) but execution continues.

- **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:
    \[\text{BI,context:}{\HELLO_1}{\HELLO_C},when:{status>0} \#15\]
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:

BI, when: {HELLO_1\HELLO_C\status>0} \HELLO_1\HELLO_C\#15

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.

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 ;.

Triggering of the breakpoint results in message being output. Prefixing message with $n$ enables you to write the message text to custom window n, where n is between 50-1024.

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.

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

Specifies the number of times that the specified condition has to occur to trigger the breakpoint. The default value is 1. 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. Only 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.

Not supported in this release.
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 just 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 will keep a log of times for each break.

update:{{@}}

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.

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\#449.3  
Set a breakpoint at line 449, statement 3 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 TRUE, the debugger continues application execution.
```

```
BI,when:{count<4 || err==5} \MAIN_1\SUBFN_C\#42  
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 FALSE, the debugger continues application 
execution.
```

Alias

BREAKINSTRUCTION and BREAK are aliases of BREAKINSTRUCTION.

See also

The following commands provide similar or related functionality:

- BREAKACCESS on page 2-40
- BREAKEXECUTION on page 2-47
- CLEARBREAK on page 2-79
- MACRO on page 2-173.
2.2.17 BREAKREAD

The BREAKREAD command sets a read breakpoint at the specified memory location(s).

Syntax

BREAKREAD [address | address_range] [;macro_call]

where:

address | address_range
        Specifies a single address in target memory, or an address range.

macro_call Specifies a macro and any parameters it requires.

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 2-119,
listing the current breakpoints.

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.

If specified, this macro runs when the breakpoint is triggered and after the instruction
at the breakpoint is executed. If the macro returns a true (nonzero) value, execution
continues. If the macro returns a false (zero) value, or if you do not specify a macro,
target execution stops and the debugger waits in command mode.

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 not allowed for a specific processor or vehicle, but this is
determined when you issue the command. 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.

continue Any triggering of the breakpoint is recorded (for example, as a
        journal entry, or by the action of a macro) but execution then
        continues.
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:{\HELLO_1\HELLO_C},when:{status>0} \HELLO_1\HELLO_C\#15
```

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:

```
BREAD,when:{\HELLO_1\HELLO_C\status>0} \HELLO_1\HELLO_C\#15
```

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.

hw_ahigh:()  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:()  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:()  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:0x440 0x1FA00
```

hw_dhigh:()  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 0x400-0x4F0:

```
BREAKREAD, hw_dvalue:0x440, hw_dmask:0xF0F 0x1FA00
```

**hw_passcount:** \((n)\)

Specifies the number of times that the specified condition has to occur to trigger the breakpoint. The default value is 1. 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 supplement the hardware one for higher count values. When both hardware and software pass counts are defined:

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 tied, 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 \textit{id} is one of:
- the breakpoint list index of an existing breakpoint
- \texttt{prev} for the last breakpoint specified for this connection.

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

\textbf{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.

\textbf{hw\_out:}\{s\}

Not supported in this release.

\textbf{hw\_not:}\{s\}

Use this qualifier to invert the sense of an address, data, or term specified in the same command. The argument \textit{s} can be set to:
- \texttt{addr} Invert the breakpoint address value.
- \texttt{data} Invert the breakpoint value.
- \texttt{then} Invert an associated \texttt{hw\_and:}\{\texttt{then}\} condition.

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

\texttt{BREAKREAD, hw\_not: data, hw\_dmask: 0x00FF ...}

The break commands require an address value, and the \texttt{addr} variant of \texttt{hw\_not} uses this address.

\texttt{BREAKREAD, hw\_not: addr 0x10040}

This means to break at any address other than \texttt{0x10040}. This is probably not useful.

The \texttt{hw\_not:then} variant of the command is used in conjunction with \texttt{hw\_and} to form \texttt{nand} and \texttt{nand-then} conditions.

This facility is not supported by ARM EmbeddedICE macrocells.

\textbf{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.

\textbf{message:}\{message\}

Triggering of the breakpoint results in \texttt{message} being output. Prefixing \texttt{message} with \texttt{$n$} enables you to write the message text to window \texttt{n}, where \texttt{n} is between 50-1024.
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.

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 specified condition has to occur to trigger the breakpoint. The default value is 1. 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. Only 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.

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:

ba, register: PR1

or

ba, register: @PR1

--- Note ---

You cannot specify core registers with this qualifier.

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 just 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 will keep a log of times for each break.

**update:**[@]

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.

**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:

- **BREAKEXECUTION** on page 2-47
- **BREAKINSTRUCTION** on page 2-55
- **BREAKREAD** on page 2-60
- **BREAKWRITE** on page 2-67
- **CLEARBREAK** on page 2-79
RealView Debugger Commands

- `DTBREAK` on page 2-119
- `ENABLEBREAK` on page 2-134.
2.2.18 BREAKWRITE

The BREAKWRITE command sets a write breakpoint at the specified memory location(s).

Syntax

```
BREAKWRITE [address | address_range][; macro_call]
```

where:

- `address | address_range` Specifies a single address in target memory, or an address range.
- `macro_call` Specifies a macro and any parameters it requires.

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 DTRBREAK on page 2-119, listing the current breakpoints.

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.

If specified, the macro runs when the breakpoint is triggered and after the instruction at the breakpoint is executed. If the macro returns a true (nonzero) value, execution continues. If the macro returns a false (zero) value, or if you do not specify a macro, target execution stops and the debugger waits in command mode.

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 not allowed for a specific processor or vehicle, but this is determined when you issue the command. 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.
- `continue` Any triggering of the breakpoint is recorded (for example, as a journal entry, or by the action of a macro) but execution then continues.
### RealView Debugger Commands

- **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:{\HELLO_1\HELLO_C},when:{status>0} #15
  ``
  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:
  ```
  BWRITE,when:{\HELLO_1\HELLO_C\status>0} \HELLO_1\HELLO_C\#15
  ```

- **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.

- **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:0x440 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:

\[ \text{BREAKWRITE, hw_dvalue:0x0, hw_dhigh:0x18} \ 0x1000 \]

**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 0x400-0x4F0:

\[ \text{BREAKWRITE, hw_dvalue:0x440, hw_dmask:0xF0F} \ 0x1FA00 \]

**hw_passcount:**(n)

Specifies the number of times that the specified condition has to occur to trigger the breakpoint. The default value is 1. 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 supplement the hardware one for higher count values. When both hardware and software pass counts are defined:

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 tied, 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 \textit{id} is one of:
- the breakpoint list index of an existing breakpoint
- \texttt{prev} for the last breakpoint specified for this connection.

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

\textbf{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.

\textbf{hw_out:}\{s\}

Not supported in this release.

\textbf{hw_not:}\{s\}

Use this qualifier to invert the sense of an address, data, or \texttt{hw_and} term specified in the same command. The argument \texttt{s} can be set to:
- \texttt{addr} Invert the breakpoint address value.
- \texttt{data} Invert the breakpoint value.
- \texttt{then} Invert an associated \texttt{hw_and:}\{\texttt{then}\} condition.

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

\texttt{BREAKWRITE, \texttt{hw_not: data, \texttt{hw_dmask: 0x00FF} ...}

The break commands require an address value, and the \texttt{addr} variant of \texttt{hw_not} uses this address.

\texttt{BREAKWRITE, \texttt{hw_not: addr 0x10040}

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

The \texttt{hw_not:then} variant of the command is used in conjunction with \texttt{hw_and} to form \texttt{nand} and \texttt{nand-then} conditions.

This facility is not supported by ARM EmbeddedICE macrocells.

\textbf{macro:}\{(\texttt{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.

\textbf{message:}\{\texttt{message}\}

Triggering of the breakpoint results in \texttt{message} being output. Prefixing \texttt{message} with $n$ enables you to write the message text to window \texttt{n}, where \texttt{n} is between 50-1024.
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.

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 specified condition has to occur to trigger the breakpoint. The default value is 1. 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. Only 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:

    ba,register:PR1

or

    ba,register:@PR1

--- Note ---

You cannot specify core registers with this qualifier.

---

sample  Not supported in this release.

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 just 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 will keep a log of times for each break.

update:{@}  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.

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 0x100..0x200  Stop program execution if the program writes to the 257 bytes from 0x100-0x200 (inclusive).

BWRITE 0x100..0x200 ; CheckMem(0x100)  Stop program execution if the program writes to the 257 bytes from 0x100-0x200 (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:

- BREAKEXECUTION on page 2-47
- BREAKINSTRUCTION on page 2-55
- \textit{BREAKREAD} on page 2-60
- \textit{BREAKWRITE} on page 2-67
- \textit{CLEARBREAK} on page 2-79
- \textit{DTBREAK} on page 2-119
- \textit{ENABLEBREAK} on page 2-134.
2.2.19 BROWSE

The BROWSE command invokes the C++ class browser interface

Syntax

BROWSE [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
```

```
parents          Shakespeare           children
-----------------|--------------------------
/                  /                     baseclass
```

See also

There are no other commands that provide similar or related functionality.
2.2.20  BWRITE

BWRITE is an alias of BREAKWRITE (see page 2-67).
2.2.21 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 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. Pended commands can be cleared from the list, and so never executed, with the CANCEL command.

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 2-158
• WAIT on page 2-297.
2.2.22 CEXPRESSION

The CEXPRESSION command calculates and displays the value of an expression. You can also modify variables using the assignment operator.

Syntax

CEXPRESSION expression

where:

expression is 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 About the CLI on page 1-2, 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 Run_Index=50
```

Assigns a value of `50` to the variable named `Run_Index`.

```
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`.

**See also**

The following commands provide similar or related functionality:

- `ADD` on page 2-17
- `DEFINE` on page 2-97
- `DUMP` on page 2-126
- `MACRO` on page 2-173
- `PRINTVALUE` on page 2-200
- `SETMEM` on page 2-221
- `SETREG` on page 2-223.
2.2.23 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 define breakpoints using the command DTBREAK (see page 2-119), 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 2-40
- BREAKEXECUTION on page 2-47
- BREAKINSTRUCTION on page 2-55
- BREAKREAD on page 2-60
- BREAKWRITE on page 2-67
- DISABLEBREAK on page 2-107
- DTBREAK on page 2-119
- ENABLEBREAK on page 2-134.
### 2.2.24 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 (..).

- `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 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 `0x88FF`, 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 $0x87FF$, 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 $0x87FF$, displaying all the differences found.

See also

The following commands provide similar or related functionality:

- *COPY* on page 2-91
- *FILL* on page 2-144
- *MEMWINDOW* on page 2-178
- *TEST* on page 2-246
- *VERIFYFILE* on page 2-290.
2.2.25 CONNECT

The CONNECT command is used to connect the debugger to a specified target.

Syntax

\texttt{CONNECT \{route\},reset|noreset \{halt|nohalt\} \{=\}[targetid]@targetname}

where:

\texttt{route} Indicates that the specified \textit{targetid} is a target access-provider, not the final target device.

\texttt{reset} Reset the target before connecting to it.

\texttt{noreset} Do not reset the target on connecting to it.

\texttt{halt} Stop the target on connecting to it.

\texttt{nohalt} Do not stop the target on connecting to it.

\texttt{targetid} Specifies the required target as a number. See Description for details.

\texttt{targetname} Specifies the required target as a name. See Description for 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 2-1 on page 2-84). However, it is helpful to think of that window when considering the operation of the CONNECT command.
Making numbered connections

The connection numbers are positive integers that identify elements in the board file. Numbers start at 1 and increment sequentially. However, it is important to understand how these numbers are allocated (to use the CONNECT command).

A distinction is made between access-provider connections and endpoint connections. For example, an access-provider connection might be an ARMulator or Multi-ICE interface unit while an endpoint connection might use ARMulator to connect to an ARM7TDMI processor, shown in Figure 2-1.

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 ids 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 2-1:

- ARMulator, access-provider connection id 1
- Multi-ICE, access-provider connection id 2
- Remote_A, access-provider connection id 3
- ARMOAK_MICE, access-provider connection id 4
- localhost Simulator Broker, access-provider connection id 5.

To connect to a Multi-ICE target accessing two ARM processors, you must first enumerate the Multi-ICE targets by connecting to the Multi-ICE access-provider:

`connect,route 2`
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.)

Typing this command the first time revises the endpoint connection ids, adding two new entries:

- ARMulator, access-provider connection id 1
- Multi-ICE, access-provider connection id 2
  - ARM7TDMI_0, endpoint connection id 6
  - ARM9TDMI_1, endpoint connection id 7
- Remote_A, access-provider connection id 3
- ARMOAK_MICE, access-provider connection id 4
- localhost Simulator Broker, access-provider connection id 5.

To connect to the first available target using Multi-ICE use CONNECT again, without the route qualifier:

```
connect 6
```

The 6 here is calculated from the number of access-provider targets plus the number of endpoint targets that use access-providers appearing before Multi-ICE in the board file.

In this case, there are 5 access-provider targets (ARMulator, Multi-ICE, Remote_A, ARMOAK_MICE and Simulator Broker). We have no other endpoint targets that appear before Multi-ICE (the only candidate is ARMulator, but you must use `connect,route 1` to enable it), so the connect id is 5 (access-providers) plus 1 (the new endpoint target) = 6.

If you then decide to open a connection to a Simulator Broker target, leaving the Multi-ICE targets connected, you can use:

```
connect,route 5
```

This revises the list, opening the Simulator Broker and adding new entries (for example, `new_OAK`):

- ARMulator, access-provider connection id 1
- Multi-ICE, access-provider connection id 2
  - ARM7TDMI_0, endpoint connection id 6
  - ARM9TDMI_1, endpoint connection id 7
- Remote_A, access-provider connection id 3
- ARMOAK_MICE, access-provider connection id 4
- localhost Simulator Broker, access-provider connection id 5
  - new_OAK, endpoint connection id 8.
To connect to new_OAK use CONNECT again, without the route qualifier:

```
connect 8
```

If you use the `connect, route` command on an access-provider that is enabled, that is expanded in the Connection Control window, this immediately hides any endpoint targets provided by that access-provider. Any endpoint connection ids allocated to it are then reused for other targets. If there are no other targets available, these ids become invalid. This means that, in the example above, you could not use the command sequence:

```
connect, route 2
connect, route 2
connect, route 5
connect 8
```

to connect to the new_OAK as this connection id would be no longer valid.

If you use the `connect, route` command on a connected target, it has no effect.

### Making named connections

You can use named connections using similar principles to the numbered technique described in *Making numbered connections* on page 2-84. 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* on page 2-87 for details).

To enable or disable a route, you enter the route name with an @ prefix. So:

```
connect, route @ARMulator
```

is (for the default board file) the same as:

```
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.)

Specify the route name as defined in the board file, that is as it appears in the Connection Control window, as shown in these examples:

```
connect, route @Multi-ICE
connect, route @Remote_A
connect, route @localhost
```
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 @ARM7TDMI_0
```

Where there are two target processors, the debugger connects to the first connection id. This depends on the order in which the access-providers were enabled.

Making named endpoint connections

You can use named connections to connect to a named target where the access-provider is not currently enabled. Specify the full target name in prefix notation:

```
connect @ARM7TDMI_0@ARMulator
```

This command connects to the ARMulator and opens the ARM7TDMI_0 device, expecting this to be available when the ARMulator is enabled. If the access-provider, in this case ARMulator, has not been configured with an ARM7TDMI_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.

Use named connections to connect to a specified target where the connection ids are unknown. For example, suppose that your targets include:

- ARMulator, configured as:
  - ARM7TDMI_0.
- Multi-ICE, configured as:
  - ARM7TDMI_0
  - ARM940T_1.

Specify the target as defined in the board file, that is as it appears in the Connection Control window. To connect to the Multi-ICE target use:

```
connect @ARM7TDMI_0@Multi-ICE
```

See also

The following commands provide similar or related functionality:

- `ADDBOARD` on page 2-20
- `BOARD` on page 2-37
- `DISCONNECT` on page 2-111
- `EDITBOARDFILE` on page 2-131
RealView Debugger Commands

- **RESTART** on page 2-213
- **RUN** on page 2-215.
2.2.26 CONTEXT

The CONTEXT command displays the current context in the Output pane.

Syntax

```
CONTEXT [/F]
```

where:

/F Displays all contexts (roots).

Description

The CONTEXT command displays the current context 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 next example shows the user setting a breakpoint in main() and running to it.

```
> 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 adds in the Root: specification shown below. See Chapter 1 Working with the CLI for more information on root context specifications.
> CONTEXT/F
At the PC: (0x0000084D0): DHRY_1_1\DHRY_1_C\main Line 98
Root: @dhrystone\ [SCOPE]

See also

The following commands provide similar or related functionality:

- `CONTEXT` on page 2-89
- `DOWN` on page 2-115
- `PRINTSYMBOLS` on page 2-195
- `SCOPE` on page 2-216
- `SETREG` on page 2-223
- `UP` on page 2-287.
2.2.27 COPY

The COPY command copies a region of memory.

Syntax

\texttt{COPY addressrange, targetaddr}

where:

\textit{addressrange} Specifies the address range to be copied.

\textit{targetaddr} Specifies the starting address where the copied memory will be placed.

Description

The \texttt{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.

\texttt{COPY} runs synchronously unless background access to target memory is supported. Use the \texttt{WAIT} command to force it to run synchronously.

Examples

The following examples show how to use \texttt{COPY}:

\begin{verbatim}
  copy 0x8100..0x81FF,0x8700
  Copies the contents of memory at 0x8100 to 0x81FF to memory at 0x8700 to 0x87FF.

  copy 0x8100..+128,0x8700
  Copies the contents of memory at 0x8100 to 0x81FF to memory at 0x8700 to 0x87FF.
\end{verbatim}

See also

The following commands provide similar or related functionality:

- \texttt{COMPARE} on page 2-81
- \texttt{FILL} on page 2-144
- \texttt{LOAD} on page 2-168
- \texttt{READFILE} on page 2-205
RealView Debugger Commands

- *TEST* on page 2-246
- *SETMEM* on page 2-221.
2.2.28 DBOARD

DBOARD is an alias of DTBOARD (see page 2-118).

2.2.29 DBREAK

DBREAK is an alias of DTBREAK (see page 2-119).
2.2.30 DCOMMANDS

The DCOMMANDS command lists the commands available based on vehicle, target processor, and type of connection.

Syntax

```
DCOMMANDS [,full | ,alias] [,cmd_class...] [;windowid]
DCOMMANDS [,full | ,alias] =specific_cmd [;windowid]
```

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` Identifies the window in which you want the command to display its output. If you do not supply a ;windowid parameter, output is displayed in the Output pane.

For further information see VOPEN on page 2-294.

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] [;windowid]
or dhelp [{,cmd_classes...}] [=specific_cmd] [;windowid]
or dtboard [=resource,...] [;windowid]
or dboard [=resource,...] [;windowid]
or dtprocess [=task,...] [;windowid]
or dvprocess [=task,...] [;windowid]
or dtfile [=value,...] [;windowid]
or dvfile [=value,...] [;windowid]
or dmap [=value,...] [;windowid]
or dtbreak [=threads,...] [;windowid]
or dbreak [=threads,...] [;windowid]
```

This command displays a more complete summary of the XTRIGGER command:

```
> dcom,full xtrig
  xtrigger [{,qualifier...}] [{boards,...}]
```

Qualifiers:
- in_disable
- in_enable
- out_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 2-159
RealView Debugger Commands

- `SHOW` on page 2-230.
2.2.31 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 Debugger User Guide for more information.
- Write your macro definition in a text file and load it into RealView Debugger using the INCLUDE command.

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:

```c
define float square(f)
    float f;
    {
        return (f*f);
    }
.

define show_i()
    {
```


fprintf 100, "value of i = %d\n", i;
return (1);
}

See also

The following commands provide similar or related functionality:

- ALIAS on page 2-23
- BGLOBAL on page 2-33
- BREAKEXECUTION on page 2-47
- CEXPRESSION on page 2-77
- GOSTEP on page 2-156
- INCLUDE on page 2-161
- MACRO on page 2-173.
- SHOW on page 2-230
- VMACRO on page 2-292.
2.2.32 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 non-connected board entry or all non-connected board entries. 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 2-83 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 2-20
- `BOARD` on page 2-37
- `CONNECT` on page 2-83
- `EDITBOARDFILE` on page 2-131.
2.2.33 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 2-17
- ALIAS on page 2-23
- PRINTSYMBOLS on page 2-195
- DEFINE on page 2-97.
2.2.34 DELFILE

The DELFILE command removes filenames from the executable file list, provided the specified file is not loaded onto the target.

Syntax

DELFILE [,auto] [name | id]

where:

auto Causes the command to remove unloaded files from the file list that were added as a result of the ADDFILE, auto command.

name|id Identifies a filename to be removed.

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.
A 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 2-121 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 2-21
- `DTFILE` on page 2-121
- `LOAD` on page 2-168
- `RELOAD` on page 2-208
- `UNLOAD` on page 2-285.
2.2.35 DHELP

DHELP is an alias of DCOMMANDS (see page 2-94).
2.2.36  **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 2-119).
- `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 reset 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 2-40
- `BREAKEXECUTION` on page 2-47
- `BREAKINSTRUCTION` on page 2-55
- `BREAKREAD` on page 2-60
- `BREAKWRITE` on page 2-67
RealView Debugger Commands

- `CLEARBREAK` on page 2-79
- `DTBREAK` on page 2-119
- `CLEARBREAK` on page 2-79
- `ENABLEBREAK` on page 2-134.
2.2.37 DISASSEMBLE

The DISASSEMBLE command displays memory addresses and corresponding assembly code on the DSM tabbed page of the Code window.

Syntax

```
DISASSEMBLE [/D|/S|/A|/J] [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.

/J Disassemble using the Java instruction set 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 tabbed 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:

- `DUMP` on page 2-126
- `LOAD` on page 2-168
- `MEMWINDOW` on page 2-178
- `MODE` on page 2-180
- `WHERE` on page 2-299.
2.2.38 DISCONNECT

The DISCONNECT command disconnects the debugger from a target.

Syntax

DISCONNECT [,all | ,gui] [=][targetid][@targetname]

where:

all  Disconnects all connections.
gui  Creates a dialog that enables you to specify the disconnect mode. This specifies what state you want the debugger to leave the target in after the disconnection. See Description 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.

When you disconnect from a target, the disconnect mode determines what happens to the target:

As-is now Leave the target in the current state. That is, if the target is running now, leave it running. If the target is stopped in debug state now, leave it stopped.

Running Leave the target running.

Stopped Leave the target stopped.

The DISCONNECT command runs asynchronously.

For more information see:

•  CONNECT on page 2-83 for details about connection ids and named connections
•  RealView Debugger v1.6 Target Configuration Guide for details on disconnect mode.
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.
- `disconnect @ARM7TDMI_0@ARMulator`  Disconnect the named ARMulator target. The `@ARMulator` is optional where there is no ambiguity. Target names must be entered as they appear in the Connection Control window.
- `disconnect @SimOAK_3@localhost`  Disconnect the named OAK target. The `@localhost` is optional where there is no ambiguity. Target names must be entered as they appear in the Connection Control window.

See also

The following commands provide similar or related functionality:

- `ADDBOARD` on page 2-20
- `CONNECT` on page 2-83
- `RESTART` on page 2-213.
2.2.39 DLOADERR

The DLOADERR command displays possible reasons for the last load error.

Syntax

dloaderr [,gui | ;windowid]

where:

gui This qualifier causes the results to be displayed in a dialog.

windowid This parameter identifies the window in which you want the command to
display its output.

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 in
the Output pane. For further information on redirecting message output see VOPEN on
page 2-294.

See also

The following commands provide similar or related functionality:

- LOAD on page 2-168
- RELOAD on page 2-208.
2.2.40 DMAP

DMAP is an alias of DTFILE (see page 2-121).
2.2.41 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 further. 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. Once `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 2-77
- CONTEXT on page 2-89
- EXPAND on page 2-140
- SCOPE on page 2-216
- UP on page 2-287
- WHERE on page 2-299.
2.2.42 DPIPEVIEW

The DPIPEVIEW command shows details of the processor execution pipe-line.

Note

This command is only supported by some types of simulator. See your simulator documentation for more information.

Syntax

DPIPEVIEW [=value] [;windowid]

where:

value A pipe level or position index which, if specified, provides more detail about the instruction at that point.

windowid Identifies the window in which you want the command to display its output. If you do not supply a ;windowid parameter, output is displayed in the Output pane. For further information see VOPEN on page 2-294.

Description

The command shows details of the processor execution pipe-line. It is normally used with cycle-accurate simulators. The pipe list contains the main pipe-line state.

Example

The following example shows how to use DPIPEVIEW.

> dpipeview
No information on pipe available
2.2.43 DTBOARD

The DTBOARD command displays information about the current or a specified board.

Syntax

```
DTBOARD [={resource,...}] [;windowid]
```

where:

- `resource` Identifies the board that is to have its details displayed.
- `windowid` Identifies the window in which you want the command to display its output.

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, output is displayed in the Output pane. For further information see `VOPEN` on page 2-294.

Example

The following example shows how to use DTBOARD.

```
> dtboard
Connected Board 'SimARM_1' Port 0: Server supporting Single Tasking.
Port string: localhost
Entry of router/broker localhost
```

Alias

DBOARD is an alias of DTBOARD.

See also

The following commands provide similar or related functionality:

- `ADDBOARD` on page 2-20
- `BOARD` on page 2-37
- `DTFILE` on page 2-121
- `DTPROCESS` on page 2-123.
2.2.44 DTBREAK

The DTBREAK command displays breakpoint information.

Syntax

DTBREAK [{thread, ...}] [{windowid}]

where:

- thread
  Not supported in this release.

- windowid
  Identifies the window in which you want the command to display its output. If you do not supply a windowid parameter, output is displayed in the Output pane. For further information see VOPEN on page 2-294.

Description

The DTBREAK command displays information about the currently defined breakpoints.

Example

The following example shows how to use DTBREAK.

```bash
> dtbreak
S Type Address Count Miscellaneous
------ -------- ----- ---------------
Instr 0x24000408  0
Read  0x24000434  0
```

Alias

DBREAK is an alias of DTBREAK.

See also

The following commands provide similar or related functionality:

- BREAKACCESS on page 2-40
- BREAKEXECUTION on page 2-47
- BREAKINSTRUCTION on page 2-55
- BREAKREAD on page 2-60
- BREAKWRITE on page 2-67
- CLEARBREAK on page 2-79
- DISABLEBREAK on page 2-107
- ENABLEBREAK on page 2-134.
2.2.45 DTFILE

The DTFILE command displays information about one or more specified files or all files of the current process.

Syntax

DTFILE [=file_number,...][;windowid]

where:

file_number,...

Is a list of 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 Identifies the window in which you want the command to display its output. If you do not supply a ;windowid parameter, output is displayed in the Output pane. For further information see VOPEN on page 2-294.

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:

Fileid Used by the ADDFILE and DELFILE commands to refer to the file.

Modid An internal number used by some RTOS loader programs.

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 in order 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 just 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 separate target name is used, or if program arguments have been defined with the ARGUMENTS command, they are also shown in the output.

Example

The following example illustrates the output of DTFILE, displaying information about a loaded executable called shapes.axf.

```bash
> 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 2-21
- LOAD on page 2-168
- MEMMAP on page 2-175
- RELOAD on page 2-208.
2.2.46 DTPROCESS

The DTPROCESS command displays information about a specified process.

Note

Debugging more than one process at a time is a separately licensed feature. See the RealView Debugger v1.6 Extensions User Guide for more information.

Syntax

DTPROCESS [={process_number,...}][;windowid]

where:

process_number,...

Identifies one or more processes about which you want to see information.

windowid

Identifies the window in which you want the command to display its output. If you do not supply a ;windowid parameter, output is displayed in the Output pane. For further information see VOPEN on page 2-294.

Description

The DTPROCESS command displays information about the processes that this connection refers to. A process is one or more sequences of control that, as a group, have a single distinct address space from any other process running on a single processor. Each sequence of control within a process is referred to as a thread. A processor normally directly implements one process. An operating system can timeslice that process to implement more than one process, and more than one thread within a process.

The RealView Debugger considers the current sequence of control on a target to be process P1 unless the RTOS extension is licensed and makes other processes on the target visible.

If you do not supply the process number, information about the current process is displayed.

Example

The following example illustrates the output of DTPROCESS, displaying information about a loaded executable.
> dtprocess =1
  'dhry.axf' As 'B:\ArmSource\dhry\Debug\dhry.axf'

**Alias**

DVPROCESS is an alias of DTPROCESS.

**See also**

The following commands provide similar or related functionality:

- *THREAD* on page 2-248
- *RELOAD* on page 2-208
- *RESTART* on page 2-213.
2.2.47 DTRACE

The DTRACE command shows information on trace.

Syntax

dtrace [;windowid]

where:

windowid Identifies the window in which you want the command to display its output. If you do not supply a ;windowid parameter, output is displayed in the Output pane. For further information see VOPEN on page 2-294.

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
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 2-26
- ETM_CONFIG on page 2-137
- TRACE on page 2-250
- TRACEBUFFER on page 2-253
- TRACEDATAACCESS on page 2-263.
2.2.48 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. However, an Oak DSP addresses words of 16 bits, so /H is the default for Oak.

/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.

Description

The DUMP command displays memory contents in bytes, words or longwords as hexadecimal and ASCII characters 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
  00008000 ED EB 97 C3 DC F4 01 1C EA E3 BF E0 1A 57 65 04 ............We.
  00008010 58 C3 0F 46 92 50 21 B5 FF 63 7E A3 16 DE 84 97 X..F.P!...c~.....

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
  ------------------------------------------------
  00008020 AC FA DF 30 22 88 9A 52  FE 94 42 D7 EB 0D 52 28 ...0"..R..B...R( 00008030 69 29 EB 5D 09 46 17 33  79 2F 69 7B FD 8C 09 D9 i).].F.3y/\{....
  00008040 21 D0 5E 63 22 88 00 58  17 C4 F8 D5 BF E9 07 84 !.^c"...X....... 00008050 1E 5A 65 27 67 08 1C 90  DD 0C 4C DE 05 51 C8 31 .Ze'g......L..Q.1
  00008060 B3 4C 61 AE 8E 15 44 80  37 EF EE 07 49 20 31 5F .La....D..7...I 1_

Requesting a DUMP of words of memory, and specifying a range of addresses produces the following result:

dump /h 0x9004..0x9012
  00009000 801F 766A 86C1 B780 2BC1 36FC ..jv......+.6
  00009010 FF8E 7ED6

See also

The following commands provide similar or related functionality:

- CEXPRESSION on page 2-77
- FILL on page 2-144
- MEMWINDOW on page 2-178
- WRITEFILE on page 2-301.
2.2.49 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.

name Specifies the filename or file pathname to which the map is written.

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 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"

This 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 2-175.
2.2.50 DVFILE

DVFILE is an alias of DTFILE (see page 2-121).

2.2.51 DVPROCESS

DVPROCESS is an alias of DTPROCESS (see page 2-123).
2.2.52 EDITBOARDFILE

The EDITBOARDFILE command enables you to edit a specified board file.

Syntax

EDITBOARDFILE [=boardfilename,...]

where:

boardfilename,... Identifies one or more board files that you want to edit.

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. 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.

editboardfile

See also

The following commands provide similar or related functionality:

- ADDBOARD on page 2-20
- DTBOARD on page 2-118
- READBOARDFILE on page 2-204.
2.2.53 EMURESET

The EMURESET command tests and resets a hardware emulator.

**Syntax**

```plaintext
EMURESET ,test
```

where:

test Runs an emulation test. This can involve JTAG testing or self checks.

**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.

The EMURESET command is not supported for ARM RDI targets. The command runs synchronously, and is not be accepted when the debugger is connected to this hardware.

**Alias**

EMURST and HWRESET are aliases of EMURESET.

**See also**

The following commands provide similar or related functionality:

- `RESET` on page 2-210
- `RESTART` on page 2-213
- `WARMSTART` on page 2-298.
2.2.54 EMURST

EMURST is an alias of EMURESET (see page 2-132).
2.2.55 ENABLEBREAK

The ENABLEBREAK command enables one or more specified breakpoints.

Syntax

```
ENABLEBREAK [,h] [\{break_num,\ldots\}]
```

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 2-119).
- `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 2-47
- `BREAKINSTRUCTION` on page 2-55
- `BREAKREAD` on page 2-60
- `BREAKWRITE` on page 2-67
- `CLEARBREAK` on page 2-79
- `DISABLEBREAK` on page 2-107
- `DTBREAK` on page 2-119
- `RESETBREAKS` on page 2-211.
2.2.56  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 2-161
• QUIT on page 2-203.
2.2.57 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 Description.

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 describing tracing in RealView Debugger v1.6 Extensions User Guide.

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 not allowed 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.

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.

fulltrace

Trace both data and address bus transfers.

Examples

The following examples show how to use ETM_CONFIG:

ETM_CONFIG port_width:4,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:8, 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 2-26
- `TRACE` on page 2-250
- `TRACEBUFFER` on page 2-253.
2.2.58 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]
```

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  Indicates that the output is to be directed to the specified window or to the file associated with that window number. You must use one of the following window numbers:

- 1  Code window (and journal file if enabled).
- 20  Standard I/O window
- 28  Log file
- 29  Journal file.
- 50–1024  Window number. For further information see `FOPEN` on page 2-149 or `VOPEN` on page 2-294.

**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\xFF\xFF\extField1E"
      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 2-77
- JOURNAL on page 2-163
- PRINTVALUE on page 2-200
- WHERE on page 2-299.
2.2.59 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 2-136
- FAILINC
- INCLUDE on page 2-161
- JUMP on page 2-165.
2.2.60  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. However, an Oak DSP addresses words of 16 bits, so /H is the default for Oak.

/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 whose memory contents will 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.

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 will be 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 h,e,l,l,o, to locations 0x1000...0x1004.

fill 0x1000..0x1001="hello"
    Writes h,e,l,l,o, to 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 h,e,l,l,o,h,e,l,l,o,... to locations 0x1000...0x1013.

fill /w 0x2032..0x2053=0xDEADC0DE
    For a little-endian memory system, writes 0xDE to 0x2032, 0xC0 to 0x2033, 0xDC to 0x2034 and on to: 0xDE to 0x2052, 0xC0 to 0x2053.
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 2-77
• MEMWINDOW on page 2-178
• SETMEM on page 2-221.
2.2.61  FLASH

The FLASH command enables you to write or erase flash blocks that have been opened.

Syntax

\[ \text{flash [{}, \text{qualifier...}]} \{, \text{address, ...}\} \]

where:

- **qualifier** If specified, must be one 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.
  - `write` Write data to the specified blocks of flash memory.
  - `verify` If you specify this qualifier the data written to the flash blocks is verified against the data source.
  - `useorig` This qualifier specifies that the original contents of the memory is used wherever it is not explicitly modified.
  - `ram:addr` If `ram` and `ramlen` are specified, they specify the region of memory used for data buffers and the flash programming code on target.
  - `ramlen:len` See `ram`. Specifies the length of the memory area that is used to store the flash programming code stored in the FME file.
  - `scratch` This qualifier specifies that the original contents of the memory buffer is not saved first. This might saves you some time if the buffer is large. By default the memory buffer is saved first, and restored afterwards.
  - `gui` Display any messages in the Flash Control dialog, not in the CLI pane of the Code window.

- **address** The flash block can be specified by address.
**Description**

This command is used to manage flash memory. This command enables you to write, erase and verify flash memory blocks that have been opened. The flash block is specified by address. You cannot program more than one target device at a time.

Debugger internal handle numbers are not available to users to identify memory blocks.

If this command is used with no arguments, it reports the currently open blocks.

**See also**

The following command provides similar or related functionality:

- *MEMMAP* on page 2-175.
2.2.62 FOPEN

The FOPEN command opens a file and assigns it a window number.

Syntax

FOPEN [/A] [/R] windowid, "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. This qualifier is used only in conjunction with the fgetc() macro.

windowid  Specifies a window or file number (in the range 50–1024).

filename  Specifies the file being opened. Quotation marks are optional if the filename consists of only alphanumeric characters, slashes, or a period. However, filenames with a leading slash must be in double quotes, and filenames with a leading backslash must be in single quotes, for example "/file" or '\file'.

Description

This command enables you to read or write a file on the host filesystem by associating it with a RealView Debugger custom window 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 2-294.

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, or by redirecting output from a command to a window with the ; specifier. You read the file using the fgetc macro. You close the file using the VCLOSE command.

The FOPEN command runs asynchronously unless in a macro.

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'
fpwrite(50, 'Some text')

Open a file and read the first character of the file.

See also

The following commands provide similar or related functionality:

- **PRINTF** on page 2-151
- **VCLOSE** on page 2-289
- **VOPEN** on page 2-294.
2.2.63  FPRINTF

The FPRINTF command displays formatted text to a specified file or window.

Syntax

PRINTF windowid, "format_string" [,argument]...

where:

windowid
    Specifies a window or file number (in the range 50–1024).

format_string
    Specifies the format to be applied to argument.

argument
    The value or values to be written.

Description

The command is similar to the C run-time fprintf function. You select the windowid to use from the range 50..1024, and it must be opened using the FOPEN command, for output to a file, or the VOPEN command, for output to a user window.

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
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 Output pane. Whenever a conversion specification is encountered, the next argument is converted according to the specification, and the result is copied to the Output pane.

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

- `PRINTF` runs synchronously
- `windowid` must be one of the predefined values or have been previously assigned by an `FOPEN` or `VOPEN` 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.

**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 2-77
-  `FOPEN` on page 2-149
-  `PRINTF` on page 2-192
-  `PRINTVALUE` on page 2-200
-  `VOPEN` on page 2-294
-  `VCLOSE` on page 2-289.
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 will continue 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.
--- 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, stopping when the current function returns to its caller.
```

See also
The following commands provide similar or related functionality:

- `BREAKEXECUTION` on page 2-47
- `BREAKINSTRUCTION` on page 2-55
- `HALT` on page 2-158
- `GOSTEP` on page 2-156
- `RUN` on page 2-215
- `STEPINSTR` on page 2-233
- `STEPLINE` on page 2-235
- `STOP` on page 2-243.
2.2.65 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 due to 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 2-47
- `BREAKINSTRUCTION` on page 2-55
- `HALT` on page 2-158
- `GO` on page 2-154
- `MODE` on page 2-180
- `RUN` on page 2-215
- `STEPINSTR` on page 2-233
- `STEPLINE` on page 2-235
- `STOP` on page 2-243.
2.2.66 HALT

The HALT command stops target program execution.

Syntax

HALT

Description

The HALT command stops the target program if it was executing. All processes or threads are stopped, and if the RTOS extension is loaded the state of any attached threads is updated.

See also

The following commands provide similar or related functionality:

- GOSTEP on page 2-156
- RUN on page 2-215
- STEPINSTR on page 2-233.
2.2.67 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.
2.2.68 HWRESET

HWRESET is an alias of EMURESET (see page 2-132).
2.2.69 INCLUDE

The INCLUDE command executes commands stored in the specified file.

Syntax

INCLUDE "filename"

where:

filename Specifies the command file to be read.

Quotation marks are optional if the filename contains only alphanumeric characters and periods. Filenames that contain a leading slash must be in double quotation marks ("/filename", for example). Filenames that contain a backslash must be in single quotation marks (\filename', for example).

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.

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. See the RealView Debugger Target Configuration Guide for more information.

You can also run script files using the -inc argument to RealView Debugger itself. See Using the CLI on page 1-6 for more information.
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 2-23
- **DEFINE** on page 2-97
- **ERROR** on page 2-136
- **FAILINC** on page 2-143
- **JUMP** on page 2-165
- **MACRO** on page 2-173.
2.2.70 JOURNAL

The **JOURNAL** command controls the logging of information displayed in the **Cmd** tab of the Output pane.

**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 if the filename contains only alphanumeric characters or periods. Filenames that contain a leading forward slash must be in double quotation marks (`"filename"`, for example). Filenames that contain a leading backward slash must be in single quotation marks (`'\\filename'`, for example).

**Description**

The **JOURNAL** command starts or stops saving, in a specified file, information displayed in the **Cmd** tab of the Output pane. The information you can direct to a file includes user command input, resulting output, error messages, and text specifically sent to the journal file.

--- **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 2-171
- `VOPEN` on page 2-294.
The JUMP command goes to a label in an include file.

**Syntax**

```
JUMP 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 a macro or 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.

If you use the JUMP command in a macro, the target label must be in the same macro.

**Example**

The following fragment of an include file shows the use of labels and jumps:

```
 initialise
 :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 2-97
- `FAILINC` on page 2-143
- The IF, WHILE and DO flow control constructs of a macro
- The macros chapter in the *RealView Debugger v1.6 User Guide*. 


2.2.72 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 will be 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 2-89
- DISASSEMBLE on page 2-109
- DOWN on page 2-115
- EXPAND on page 2-140
- SCOPE on page 2-216
- UP on page 2-287
- WHERE on page 2-299.
2.2.73 LOAD

The LOAD command loads the specified executable file into the debugger.

Syntax

```
```

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.

/NW
The option does nothing in this release of RealView Debugger.

absolute_filename
Specifies the name of the absolute object file to be loaded. Quotation marks are optional if the filename contains only alphanumeric characters or periods. Filenames that contain a leading slash must be in double
quotation marks ("/filename", for example). Filenames that contain a leading backslash must be in single quotation marks ('ilename', for example).

root Specifies the root associated with the symbols in the program being loaded. The default root is the filename without an extension.

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).

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.

args Specifies an optional, space-separated, list of arguments to the image. Where several arguments are provided, single quotes must be used. The case of arguments is preserved.

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.

Examples

The following examples show how to use LOAD:

load c:\source\myfile.axf

Load the executable file myfile.axf to the target.
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 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 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 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 2-21
- GO on page 2-154
- RELOAD on page 2-208
- RESET on page 2-210
- RESTART on page 2-213
- RUN on page 2-215
- UNLOAD on page 2-285.
2.2.74 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.

ON Starts writing information to the log file.

filename Specifies the name of the log file. Quotation marks are optional if the filename contains only alphanumeric characters or periods. Filenames that contain a leading slash must be in double quotation marks ("/filename", for example). Filenames that contain a leading backslash must be in single quotation marks ('\filename', for example). If you do not supply a filename extension, the extension .log is used by default.

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.

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.

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:

\begin{verbatim}
LOG ON='c:\temp\log.txt'
\end{verbatim}
Start logging output to the file c:\temp\log.txt, overwriting any existing
file of that name.

\begin{verbatim}
LOG /A ON="log"
\end{verbatim}
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:
\begin{itemize}
\item JOURNAL on page 2-163
\item VOPEN on page 2-294.
\end{itemize}
2.2.75 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 need 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`:

```plaintext
macro fgetc(50)
    Read a character from the file associated with window 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 2-23
- `CEXPRESS` on page 2-77
- `DEFINE` on page 2-97
- `INCLUDE` on page 2-161
- `PRINTSYMBOL` on page 2-195.
- `SHOW` on page 2-230
- `VMACRO` on page 2-292.
2.2.76 MEMMAP

The MEMMAP command enables you to define and control memory mapping.

Syntax

MEMMAP [{,qualifier...}] [=address]

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.
  - **update automap** 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.
description: 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.

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.
For DSP Group Oak and TeakLite processors, the types are:
- Program
- Data
- IO.

description  The memory region, specified as a single address (delete) or an address range (define).
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 allowed 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`:

```
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:

- `MEMWINDOW` on page 2-178
- `SETMEM` on page 2-221.
2.2.77 MEMWINDOW

The MEMWINDOW command sets the base address of the memory pane.

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. However, an Oak DSP has a natural addressing of a word (16 bits), so /H is the default for Oak.

/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 2-221.
2.2.78 MMAP

MMAP is an alias of MEMMAP (see page 2-175).
2.2.79 MODE

The MODE command switches the code window between disassembly and source view.

Syntax

\texttt{MODE [HIGHLEVEL | ASSEMBLY]}

where:

\texttt{HIGHLEVEL} \hspace{1cm} Set the code window to the source view.

\texttt{ASSEMBLY} \hspace{1cm} 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 GOSSTEP 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 2-89.
- DISASSEMBLE on page 2-109.
- GOSSTEP on page 2-156
- LIST on page 2-166.
2.2.80 MONITOR

The MONITOR command adds the named variable to the list of monitored, or watched, variables, displayed in the Watch pane.

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:

- `CEXPRESS` on page 2-77
- `CONTEXT` on page 2-89
- `DUMP` on page 2-126.
- `NOMONITOR` on page 2-184
- `PRINTVALUE` on page 2-200.
2.2.81 NAMETRANSLATE

The NAMETRANSLATE command manipulates the host/target name translation list.

Syntax

\texttt{NAMETRANSLATE [{,\texttt{qualifier...}}] [{\texttt{name-translation, ...}}]}

where:

\texttt{qualifier} If specified, must be one of the following:
  \texttt{replace} The current name translation list is to be replaced by the list specified in this command.
  \texttt{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.

\texttt{name_translation}

A list of name translations. The format is:

\texttt{hname,hname,hname=tname,tname}

where \texttt{hname} is the filename on the host (or a comma-separated list of them) and \texttt{tname} is the filename on the target (or a comma-separated list of them).

The whole 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 displays the current name translation list.

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.

See also

The following commands provide similar or related functionality:

- \texttt{LOAD} on page 2-168
- \texttt{PATHTRANSLATE} on page 2-190.
2.2.82  NOMONITOR

The NOMONITOR command deletes variables from the Watch pane.

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.

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 line 2, 3, and 4 of the watch pane.

See also

The following command provides similar or related functionality:

•   MONITOR on page 2-181.
2.2.83 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 just before program execution starts.
  - **stop** Execute the command just after program execution stops.
  - **starttimed** Execute the command just 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 just 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:

```plaintext
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 0x8000
   On Reset:  
      <no commands registered>
```

See also

The following command provides similar or related functionality:

- BGLOBAL on page 2-33.
2.2.84 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. A decimal number can also be suffixed 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. A hexadecimal number can also be prefixed with 0x or suffixed with h.

    **Note**
    You must prefix every hex number with a numeric digit to avoid confusion with symbol names.

  - **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 endianess 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 endianess 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:

RADI X = 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).

See also

The following commands provide similar or related functionality:

- CEXPRESSION on page 2-77
- LOAD on page 2-168
- PRINTVALUE on page 2-200
- SETTINGS on page 2-226.
2.2.85 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.
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. The format is:
host=target
Target may be '-' which means a grab of a process with no path will prepend 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 supply no arguments, the PATHTRANSLATE command displays the current name translation list.

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.

See also

The following commands provide similar or related functionality:
• LOAD on page 2-168
• NAMETRANSLATE on page 2-183.
2.2.86 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 the user indicates 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.

**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 2-297.
2.2.87 PRINTF

The PRINTF command prints formatted text to the Output pane.

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 values are to
be presented.

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
command 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.

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
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.

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,r0,LSL r4
    Instruction2 000000104 20011412 ANDCS 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 2-77
- FPRINTF on page 2-151
- PRINTTYPE on page 2-198
- PRINTVALUE on page 2-200.
2.2.88 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. All symbols are displayed that match the name when no options are specified.

——— 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 program. For example, for the dhrystone program this command prints:

@dhry\TIME_H : Codeless Module.
@dhry\STDIO_H : Codeless Module.
@dhry\DHRY_H : Codeless Module.
@dhry\DHRY_2 : High level module.
               Code section = 0100004C to 010002BB
@dhry\DHRY_1 : High level module.
               Code section = 010002BC to 010010F7
@dhry\ENTRY : Assembly level module.
               Code section = 01000000 to 0100004B
@dhry\ARMLIB_CN_32L : Assembly level module.
                   Code section = 010010F8 to 01009423
@dhry\SYSTEM : Assembly level module.
               Code section = 01009424 to 0100945F
@dhry\CONST : Assembly level module.
               Code section = 01009460 to 01009477
@dhry\DATA : Assembly level module.
               Code section = 01009478 to 0100962B
@dhry\BSS : Codeless Module.
@dhry\UNNAMED_1 : Codeless Module.
Alias

PS is an alias of PRINTSYMBOLS.

See also

The following command provides similar or related functionality:

- PRINTTYPE on page 2-198.
2.2.89 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 program:

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 2-17
- **BROWSE** on page 2-74
- **DELETE** on page 2-101
- **PRINTF** on page 2-192
- **PRINTSYMBOLS** on page 2-195.
2.2.90 PRINTVALUE

The PRINTVALUE command prints the value of a variable or expression.

Syntax

PRINTVALUE [/H|/S|/T] {expression | expression_range}

where:

/T Displays the value in decimal format.
/H Displays the value in hexadecimal format.
/S Displays character pointers by dereferencing the pointer and showing the result as a null-terminated string.

expression Specifies an expression to be displayed in the Output pane.
expression_range Specifies an expression range to be displayed in the Output pane.

Description

The PRINTVALUE command prints to the Output pane 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.

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 *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
0100C2A8 = {Ptr_Comp=(record *)0x0100C274,Discr=Ident_1,
variant={var_1={Enum_Comp=Ident_3,Int_Comp=17,
Str Comp="DHRYSTONE PROGRAM, SOME STRING",
Enum_Comp=Ident_3,Int_Comp=17,Str Comp="DHRYSTONE
E PROGRAM, SOME STRING",Enum_Comp=Ident_3,
Int Comp=17,Str Comp="DHRYSTONE PROGRAM, SOME ST
RING"},var_1={Enum_Comp=Ident_3,Int_Comp=17,
Str Comp="DHRYSTONE PROGRAM, SOME STRING",
Enum_Comp=Ident_3,Int Comp=17,Str Comp="DHRYSTONE
E PROGRAM, SOME STRING",Enum_Comp=Ident_3,
Int Comp=17,Str Comp="DHRYSTONE PROGRAM, SOME ST
RING"}}
```

Note

For the same expression, CEXPRESSION prints the address, not the full value:

```
> ce *Ptr_Glob
Result is: data address 0100C2A8
```

p Ptr_Glob

Printing the value of the pointer tells you the address of the pointer, its type and the value stored there:

```
01009478 = (record *)0x0100C2A8
```

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 0100C2A8
```

See also

The following commands provide similar or related functionality:

- CEXPRESSION on page 2-77
- MONITOR on page 2-181.
2.2.91 PROPERTIES

PROPERTIES is an alias of SETTINGS (see page 2-226).

2.2.92 PS

PS is an alias of PRINTSYMBOLS (see page 2-195).

2.2.93 PT

PT is an alias of PRINTTYPE (see page 2-198).
2.2.94 QUIT

The QUIT command causes the debugger to exit.

Syntax

QUIT [Y]

where:

Y    Exits the debugger without displaying a confirmation dialog.

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 further confirmation. The debugger warns of any unsaved changes.
2.2.95 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.

--- Note ---

For this command, the file path name must be enclosed in double quotes.

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 2-20
- `BOARD` on page 2-37
- `DELBOARD` on page 2-100
- `EDITBOARDFILE` on page 2-131.
2.2.96 READFILE

The READFILE command reads a file into target memory.

Syntax

```
READFILE ,[obj|raw|ascii] [,opts] {name} [=address]
```

where:

- **obj**  The file is an executable file in the standard target format. For ARM targets, this is ARM-ELF. There are no *opts* supported for this file type.

- **raw**  The file is a stream of 8-bit values that are written to target memory without further interpretation. There are no *opts* supported for this file type.

- **ascii**  The file is a stream of ASCII digits separated by whitespace. The interpretation of the digits is specified by other qualifiers. The starting address of the file must be specified in a bracketed line 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.

The following *opts* are supported for this file type:

- **byte**  The file is a stream of 8-bit values that are written to target memory without further 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.

- **name**  Specifies the filename to be read.
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.

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 and Oak processors have word 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 2-144
- LOAD on page 2-168
- SETMEM on page 2-221
- VERIFYFILE on page 2-290
- WRITEFILE on page 2-301.
2.2.97 REEXEC

REEXEC is an alias of RESTART (see page 2-213).
2.2.98 RELOAD

The **RELOAD** command loads a linked program image containing program code and data.

**Syntax**

```
RELOAD [{,qualifier...] [name | id] [=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.
- **killtasks**  Applicable only to RTOS and when threads identify the entry function. Attempts to identify any threads connected to a file being unloaded or reloaded and stop their execution.
- **nokilltasks**  Does not attempt to identify affected threads and stop their execution.

**name | id**  Specifies the file to be reloaded, either by its name or by an identifier allowing its selection from the executable file list. If you do not specify a file, the whole process is reloaded.

**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.

See also

The following commands provide similar or related functionality:

- `ADDFILE` on page 2-21
- `LOAD` on page 2-168
- `READFILE` on page 2-205
- `UNLOAD` on page 2-285.
2.2.99  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 processor that is to be reset.

**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 will vary 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 reinitialized.

The RESET command runs synchronously.

**Alias**

`WARMSTART` is an alias of `RESET`.

**See also**

The following command provides similar or related functionality:

- `RESTART` on page 2-213.
2.2.100 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 2-119).

- **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 2-119). It also resets the *and* and *and-then* condition state so that the first breakpoint is once again required before the second can trigger. For more information on *and* and *and-then* conditions see BREAKEXECUTION on page 2-47.

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:

- **BREADEXECUTION** on page 2-47
- **BREAKINSTRUCTION** on page 2-55
- **BREAKREAD** on page 2-60
- **BREAKWRITE** on page 2-67
- **CLEARBREAK** on page 2-79
- **DTBREAK** on page 2-119
- **RELOAD** on page 2-208.
2.2.101 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.

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 2-26) 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 will fail.
• The RESTART command might behave differently if you are using the RTOS extension to RealView Debugger. Refer to 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 2-154
• RELOAD on page 2-208.
2.2.102 RSTBREAKS

RSTBREAKS is an alias of RESETBREAKS (see page 2-211).
2.2.103 RUN

The RUN command starts execution using a specific mode, or sets the default mode used by the GO command.

Syntax

RUN [,.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 2-154.
2.2.104 SCOPE

The SCOPE command specifies the current module and procedure scope.

Syntax

SCOPE /F
SCOPE root_name\nSCOPE [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: (0x0100028C): 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 2-89
- PRINTVALUE on page 2-200
- WHERE on page 2-299.
2.2.105 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. However, an Oak DSP has a natural addressing of a word (16 bits), so `/H` is the default for Oak.

- `/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.

**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 2-178
- SETMEM on page 2-221.
2.2.106 SETFLAGS

The SETFLAGS command is reserved for internal use by the RealView Debugger.
2.2.107 SETMEM

The SETMEM command changes the contents of memory to a specified value.

**Syntax**

```plaintext
SETMEM [/B|/H|/W|/8|/16|/32] address [=expression | expression_string]
```

where:

- `/B`, `/8` Sets the fill size to byte (8 bits). 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).
- `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:

```c
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:

```bash
setmem /32 &count=5
setmem /32 0x10200=5
```

The following two statements both set the value of `buf[0]` to `0x40`:

```bash
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 2-77
- `FILL` on page 2-144.
2.2.108 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.

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 register names by selecting Debug → Simple Breakpoints → Simple Break if X, click on the down-arrow after the expression text field and selecting <Register list...>.

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 SETMEM.

——— 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

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 2-2. 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.

![Interactive Register Setting dialog](image)

Figure 2-2 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 2-3 on page 2-225, enabling you to enter a new value for that register.
Figure 2-3 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**

$R$ is an alias of `SETREG`.

**Examples**

The following examples show how to use `SETREG`:

```plaintext
setreg @r3=0x50

Write the value 0x50 to processor register R3.
```

```plaintext
setreg @spsr_svc

Display a prompt, shown in Figure 2-3, 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.
```

```plaintext
setreg @v=1

Set the ALU overflow flag in the current program status register.
```

```plaintext
setreg

Invoke the Interactive Register Setting dialog shown in Figure 2-2 on page 2-224.
```

**See also**

The following commands provide similar or related functionality:

- `ADD` on page 2-17
- `CEXPRESSION` on page 2-77.
2.2.109 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 *Description*.

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 are also set using the project manager interface.

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.

The standard option names are:

- **loadact**: Action on load. The possible values are:
  - `default`: Normal load image behavior. For ARM processors, the processor is placed in ARM state and supervisor mode with interrupts disabled.
  - `nomask`: 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.
**pcset** When to change the PC when loading an image. The possible values are:

- **auto** Normal 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 `/S` 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.

**pcdefault** User default PC value. This value is written to the target PC when an image file is `LOADed` or `RELOADed` and the `pcset` option is `user_default`.

**reset** 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 `SETREG` on page 2-223 for more information on possible register names.

**restart** 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.

**restart_reset** Reset on restart. The possible values are:

- **true** Reset the processor on `RESTART`, as well as any other actions.
- **false** Do not reset the processor on `RESTART`.

**verify** Verify the image `LOADed` or `RELOADed`. 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.

**full**  Read and check every byte of the image.

**verify_warns**  Determines what happens if an image file verification failure occurs. The possible values are:

- **true**  The failure is treated as a warning and operation continues.
- **false**  The failure is an error and the image load is aborted.

**fillstack**  Define a value to fill stack memory with before the program starts. This is not used for ARM processor image files.

**fillheap**  Define a value to fill memory defined as heap. This is not used for ARM processor image files.

**fillundefined**  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.

**disasm**  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 Java instructions, 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.

**dsmvalue**  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.
Additional options might be implemented for particular target interfaces. Refer to the target interface documentation for more information.

**Alias**

PROPERTIES is an alias of SETTINGS.

**Examples**

The following examples show how to use SETTINGS:

```plaintext
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 2-109
- *LOAD* on page 2-168
- *RELOAD* on page 2-208
- *OPTION* on page 2-187
- *RESET* on page 2-210
- *RESTART* on page 2-213.
2.2.110  SHOW

The SHOW command displays the source code of a specified debugger macro.

Syntax

SHOW macro_name [;windowid]

where:

macro_name Specifies the name of the macro to be displayed.

windowid Identifies the window in which the macro is to be displayed. Valid values include:

20  Standard I/O window.
28  Log file.
29  Journal File.
50–1024  Window or file number. For further information see VOPEN on page 2-294.

If you do not supply a windowid parameter, 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 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 2-97
- INCLUDE on page 2-161
- MACRO on page 2-173.
2.2.111 SIMULATOR

The SIMULATOR command configures the processor simulator.

--- Note ---
This command can only be used if the simulator supports it.

---

Syntax

```
simulator ,cyclestep
simulator (,savestate|,restorestate) ="filename"
simulator ,properties =propertyname
```

where:

cyclestep Switch between processor cycle simulation and instruction simulation

properties Set the simulator properties. There are no properties you can set for the MaxCore DSP simulator, but other simulators might provide them. See your simulator documentation for more information.

savestate Save the simulator state to the named file, for example, register contents, memory contents, in an internal buffer for later restoration with restorestate.

restorestate Restore the simulator state from the named file.

Description

The SIMULATOR command can switch between cycle stepping and instruction stepping for simulators that support this. It can also edit simulator properties, and save and restore the simulator state.

See also

The following commands provide similar or related functionality:

- EMURESET on page 2-132
- OPTION on page 2-187
- SETTINGS on page 2-226.
2.2.112 SINSTR

SINSTR is an alias of STEPINSTR (see page 2-233).

2.2.113 SM

SM is an alias of SETMEM (see page 2-221).

2.2.114 SOINSTR

SOINSTR is an alias of STEPOINSTR (see page 2-238).

2.2.115 SOVERLINE

SOVERLINE is an alias of STEPO (see page 2-240).

2.2.116 SR

SR is an alias of SETREG (see page 2-223).
2.2.117 STEPINSTR

The STEPINSTR command executes a specified number of processor instructions.

Syntax

```
STEPI
STEPI =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 true 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:

\texttt{stepinstr}
\begin{quote}
Step the program by one instruction.
\end{quote}

\texttt{si 5}
\begin{quote}
Step the program five instructions.
\end{quote}

\texttt{si =0x8000,5}
\begin{quote}
Starting at address $0x8000$, step the program five instructions.
\end{quote}

Alias

\texttt{SINSTR} is an alias of \texttt{STEPINSTR}.

See also

The following commands provide similar or related functionality:

- \texttt{BEXECUTION} on page 2-32
- \texttt{DISASSEMBLE} on page 2-109
- \texttt{GO} on page 2-154
- \texttt{GOSTEP} on page 2-156
- \texttt{MODE} on page 2-180
- \texttt{STEPLINE} on page 2-235.
2.2.118 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 2-32
- *DISASSEMBLE* on page 2-109
- *GO* on page 2-154
- *GOSTEP* on page 2-156
- *MODE* on page 2-180
- *STEPINSTR* on page 2-233.
2.2.119 STEPOINSTR

The STEPOINSTR command executes a specified number of instructions, and completely executes program calls.

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 STEPINSTR.

See also
The following commands provide similar or related functionality:
- BEXECUTION on page 2-32
- DISASSEMBLE on page 2-109
- GO on page 2-154
- GOSTEP on page 2-156
- MODE on page 2-180
- STEPINSTR on page 2-233
- STEPLINE on page 2-235
- STEPO on page 2-240.
2.2.120 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:

stepto 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 2-32
• DISASSEMBLE on page 2-109
• GO on page 2-154
• GOSTEP on page 2-156
• MODE on page 2-180
• STEPINSTR on page 2-233
RealView Debugger Commands

- `STEPLINE` on page 2-235
- `STEPINST` on page 2-238.
2.2.121 STOP

The STOP command with no argument is equivalent to HALT (see page 2-158).
2.2.122 SYNCHEXEC

The SYNCHEXEC command controls how connections, threads and boards run, step, and stop together.

Note

Debugging more than one processor at a time is a separately licensed feature. See RealView Debugger Extensions Guide for more information about the multiprocessing features of RealView Debugger.

Syntax

SYNCHEXEC [,run] [,step] [,stop] [listname] | [,all] | [ [listname=] connections]

SYNCHEXEC rem, [listname] | [,all] | [ [listname=] connections]

where:

run, step, stop

Qualifiers that you can specify in any combination to define the actions to synchronize. If you do not supply a qualifier, all are assumed together.

remove

Remove the connections from the synchronized group instead of adding them to it.

all

Indicates all currently made connections that can be synchronized.

listname

A name that you give to a connection list to simplify referring to it in later commands. You define a list by prefixing a list with a name and the assignment = operator.

connections

A comma-separated list of connection identifiers, of the form:

threadname[@boardname], threadname[@boardname],...

or

connection-id, connection-id

where:

threadname

The name of a thread on a particular board, as in the Code window title bar.

boardname

The name of a target board, as in the Code window title bar.

connection-id

A number representing a target connection, as described for CONNECT page 2-83.
Description

The SYNCHEXEC command controls how RealView Debugger controls multiple target processors. The initial state is that every target processor is controlled independently. Therefore, stopping or starting a program on one processor only affects other processors if there is a link between the processors.

If you require RealView Debugger to stop or start several target processors together, you use this command to link them into a synchronized execution group. You can choose whether this group applies to single stepping, to free-running, or to stopping (in the sense of a user-initiated halt) independently.

Examples

The following examples show how to use SYNCHEXEC:

`synchexec, rem, all`

Unsynchronize all processors in the processor group.

`synchexec, step, run, stop 7, 8, 9`

Synch endpoint connections 7, 8, and 9 on step, stop and run.

`synchexec, all`

Synchs all available endpoint connections on step, stop and run.

See also

The following commands provide similar or related functionality:

- BEXECUTION on page 2-32
- GO on page 2-154
- STEPINSTR on page 2-233
- XTRIGGER on page 2-303.
2.2.123 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.

`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 2-221.
2.2.124 THREAD

The THREAD command sets the specified thread, or process and thread, to be the current thread, or process and thread.

--- Note ---

See the RealView Debugger Extensions Guide for more information about the multiprocessor and RTOS features of the debugger.

Debugging more than one processor at a time is a separately licensed feature.

Syntax

THREAD [,next][,default]

THREAD [=task]

where:

next       Change the current thread to be the next one in the list of threads.

default    Change the selected thread to be the thread that was executing when the target application stopped.

task       Define the process or 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, or process and thread, to be the current thread, or process and thread.

The current process is normally set by the last one grabbed. The current thread is normally set by whichever thread stops last. This command enables you to specify a thread, or a thread of a specific process, 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 =2

Change the current thread to thread 2.

See also

The following commands provide similar or related functionality:

- BOARD on page 2-37
- RUN on page 2-215.
2.2.125 TRACE

The TRACE command is used to enable or disable tracing, both globally and during program execution.

Syntax

TRACE location
TRACE [,endpoint|,prompt|,trigger] location
TRACE ,range startlocation .. endlocation
TRACE [,start [,toggle]],stop

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 performs one of two tasks:

- globally enabling or disabling all trace activity.
- setting 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.

Trace operation is described in detail in the Trace chapter of the *RealView Debugger v1.6 Extensions User Guide*.

**Global trace control**

The start and stop qualifiers are used to enable or disable tracing:

- **trace ,start**: Enables tracing, given that trace has already been configured.
- **trace ,start,toggle**: Enables tracing, if currently disabled and trace has already been configured, or disables tracing if trace is currently enabled.
- **trace ,stop**: Disables tracing.
**Trace control during program execution**

The endpoint, range, 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 a tracepoint in the program at address `location`, where the type of tracepoint is selected from a list of supported types presented in a dialog.

- **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 ,range ,data startlocation .. endlocation**
  Set a trace range in the program from address `startlocation` to `endlocation`, so that instructions executed or 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 show up on data tracing, and might fill up the ETM FIFO buffer quite quickly, depending on the program.
Examples

The following examples show how to use TRACE:

TRACE, tog, start
   If tracing is currently disable it, enable it. If it is currently enabled, disable it.

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 2-26
- DTRACE on page 2-125
- ETM_CONFIG on page 2-137
- TRACEBUFFER on page 2-253
- TRACEDATAACCESS on page 2-263
- TRACEDATAREAD on page 2-268
- TRACEDATAWRITE on page 2-273
- TRACEINSTREXEC on page 2-277
- TRACEINSTRFETCH on page 2-281.
2.2.126 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 Debugger v1.6 Extensions User Guide.

Subcommands

The possible subcommands listed in the syntax are described in the following sections:

• Loadfile on page 2-254
• Savefile on page 2-254
• Closefile on page 2-255
• Amount on page 2-255
• Scaletime on page 2-256
• Speed on page 2-256
• Find_trigger on page 2-256
• Find_position on page 2-256
• Find_time on page 2-257
RealView Debugger Commands

- Find_address on page 2-257
- Find_data on page 2-257
- Find_name on page 2-257
- Posfilter on page 2-258
- Timefilter on page 2-258
- Addressfilter on page 2-258
- Namefilter on page 2-259
- Percentfilter on page 2-259
- Dvalfilter on page 2-259
- Typefilter on page 2-259
- Clearfilter on page 2-260
- Or_filter on page 2-260
- And_filter on page 2-260
- Pos_relative on page 2-260
- Pos_absolute on page 2-261
- Gui on page 2-261.

Loadfile

TRACEBUFFER ,loadfile ="filename"

Load a file into the trace buffer for further analysis. *filename* is the name of the file to load, and must be quoted.

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 again.
- 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).

**Closefile**

```
TRACEBUFFER ,closefile
```

Unload the data from the last file loaded with `loadfile` and clear the Analysis window.

**Amount**

```
TRACEBUFFER ,amount =size
```

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`.

**Scaledtime**

```
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:

\[
\begin{align*}
  n & : \text{The time to find.} \\
  n..m & : \text{Find the first in a range of times from } n \text{ to } m \text{ inclusive.} \\
  n..+o & : \text{Find the first in a range of times from } n \text{ to } n+o \text{ inclusive.}
\end{align*}
\]

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:

\[
\begin{align*}
  n & : \text{The address to find.} \\
  n..m & : \text{Find the first in a range of addresses from } n \text{ to } m \text{ inclusive.} \\
  n..+o & : \text{Find the first in a range of addresses from } n \text{ to } n+o \text{ inclusive.}
\end{align*}
\]

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:

\[
\begin{align*}
  n & : \text{The data bus value to find.} \\
  n..m & : \text{Find the first in a range of data bus values from } n \text{ to } m \text{ inclusive.} \\
  n..+o & : \text{Find the first in a range of data bus values from } n \text{ to } n+o \text{ inclusive.}
\end{align*}
\]

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 2-254 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 2-254 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 \texttt{addressfilter}. Use \texttt{namefilter} instead.

Applying a filter to the trace buffer does not lose information unless you save the trace with the \texttt{filtered} qualifier. See \textit{Savefile} on page 2-254 for more information.

\textbf{Namefilter}

\texttt{TRACEBUFFER .namefilter ="name"}

Restricts the trace buffer information displayed in the Analysis window based on a symbolic name, where \texttt{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 \texttt{filtered} qualifier. See \textit{Savefile} on page 2-254 for more information.

\textbf{Percentfilter}

\texttt{TRACEBUFFER .percentfilter =percent}

Restricts the trace buffer information displayed in the Analysis window based on a percentage of the buffer, where \texttt{percent} is an integer or range:

- \texttt{n} The percentage to display.
- \texttt{n..m} Display the range of percentages from \texttt{n} to \texttt{m} inclusive.
- \texttt{n..+o} Display the range of percentages from \texttt{n} to \texttt{n+o} inclusive.

Applying a filter to the trace buffer does not lose information unless you save the trace with the \texttt{filtered} qualifier. See \textit{Savefile} on page 2-254 for more information.

\textbf{Dvalfilter}

\texttt{TRACEBUFFER .dvalfilter =value}

\texttt{TRACEBUFFER .dvalvaluefilter =value}

Restricts the trace buffer information displayed in the Analysis window based on a data value or range of values, where \texttt{value} is an integer or range:

- \texttt{n} The value to display.
- \texttt{n..m} Display the range of value from \texttt{n} to \texttt{m} inclusive.
- \texttt{n..+o} Display the range of value from \texttt{n} to \texttt{n+o} inclusive.

Applying a filter to the trace buffer does not lose information unless you save the trace with the \texttt{filtered} qualifier. See \textit{Savefile} on page 2-254 for more information.

\textbf{Typefilter}

\texttt{TRACEBUFFER .typefilter =mask}
TRACEBUFFER, accessstypemask =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 2-254 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.

**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 to the user must use dialogs rather than text in the command window.

**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 2-26
- `ETM_CONFIG` on page 2-137
RealView Debugger Commands

- TRACE on page 2-250
- TRACEDATAACCESS on page 2-263
- TRACEDATAREAD on page 2-268
- TRACEDATAWRITE on page 2-273
- TRACEINSTREXEC on page 2-277
- TRACEINSTRFETCH on page 2-281.
2.2.127 TRACEDATAACCESS

The TRACEDATAACCESS command sets a trace point on data accesses, that is, either reads or writes.

Syntax

```
TRACEDATA ACCESS [ , qualifier ]... [ address | address_range ]
```

where:

- `qualifier` is an ordered list of zero or more qualifiers. The possible qualifiers are described in Description.
- `address` specifies the address at which the tracepoint is placed.
- `address_range` specifies the address range for the tracepoint.

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 describing tracing in RealView Debugger v1.6 Extensions User Guide.

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.
- `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 \(0x1FA00\):

\[ \text{TRCDA, hw_dvalue:0x440 0x1FA00} \]

hw_dhigh: \((n)\)  
Specifies the high data value for a data-range tracepoint. The low data value is specified by the \(\text{hw_dvalue}\) 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 \(0x1000\):

\[ \text{TRCDA, hw_dvalue:0x0, hw_dhigh:0x18 0x1000} \]

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 \(\text{hw_dvalue}\) 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 \(0x400\)-\(0x4F0\) at an instruction at address \(0x1000\):

\[ \text{TRCDA, hw_dvalue:0x440, hw_dmask:0xF0F 0x1FA00} \]

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: \{\text{[then-]}id\}\)  
Perform an \textit{and} or an \textit{and-then} conjunction with an existing tracepoint. For example, \(\text{hw_and:2,}\) or \(\text{hw_and:then-2}\), where \(2\) is the tracepoint id of another tracepoint.

In the \textit{and} form, the conditions associated with both tracepoints are tied, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the \textit{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
• 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.

For example, this command sets a tracepoint that triggered 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" 0x1E000..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:

- **Trace Point 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:"Trace Point 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.

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

### Examples

The following examples show how to use TRACEDATAACCESS:

```
TRACEDATAACCESS \MATH\#449.3
```

Set a trace trigger at statement 3 of line 449 in the file math.c.

```
TRCDA \MAIN\#35..
```

Start tracing instructions when a data access in the code between line 35-63 of main.c occurs.

```
TRCDA,hw_pass:5,hw_out:"Trace Point Type=Start Tracing" \MAIN\#35
```

Start tracing when a data access at line 35 of main.c occurs.

```
TRCDA,hw_out:"Trace Point Type=Stop Tracing" \GUI\#35..\GUI\#78
```

Stop tracing when any instruction between line 35-78 of gui.c accesses data.
Alias

TRCDACCESS is an alias of TRACEDATAACCESS.

See also

The following commands provide similar or related functionality:

- ANALYZER on page 2-26
- ETM_CONFIG on page 2-137
- TRACE on page 2-250
- TRACEBUFFER on page 2-253
- TRACEDATAREAD on page 2-268
- TRACEDATAWRITE on page 2-273
- TRACEINSTREXEC on page 2-277
- TRACEINSTRFETCH on page 2-281.
2.2.128 TRACEDATAREAD

The TRACEDATAREAD command sets a trace point on data reads.

**Syntax**

```
TRACEDATAREAD [,qualifier]... [address|address_range]
```

where:

- **qualifier** is an ordered list of zero or more qualifiers. The possible qualifiers are described in Description.
- **address** specifies the address at which the tracepoint is placed.
- **address_range** specifies the address range at which the tracepoint is placed.

**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 describing tracing in *RealView Debugger v1.6 Extensions User Guide*.

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.

- **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:

\[\text{TRCDR, hw_dvalue:0x440 0x1FA00}\]

hw_dhigh: \((n)\) Specifies the high data value for a data-range tracepoint. The low data value is specified by the \text{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:

\[\text{TRCDR, hw_dvalue:0x0, hw_dhigh:0x18 0x1000}\]

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 \text{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 0x400-0xF0 at an instruction at address 0x1000:

\[\text{TRCDR, hw_dvalue:0x440, hw_dmask:0xF0F 0x1FA00}\]

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-]}\text{id}\) Perform an \textit{and} or an \textit{and-then} conjunction with an existing tracepoint. For example, \text{hw_and:2}, or \text{hw_and:then-2}, where 2 is the tracepoint id of another tracepoint.

In the \textit{and} form, the conditions associated with both tracepoints are tied, so that the action associated with the second tracepoint is performed only when both conditions match at the same time.

In the \textit{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 \textit{id} is one of:
- the tracepoint list index of an existing of tracepoint

---

RealView Debugger Commands
RealView Debugger Commands

- **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.

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:

**Trace Point 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 0x1E00-0x1FF00, but does not trace instructions outside this range:

`TRCDR,hw_out:"Trace Point Type=Trace Instr" 0x1E00..0x1FF00`

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.

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

**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:Trace Point Type=Start Tracing \MAIN_1\#35`

Start tracing when a data read occurs at line 35 of `main.c`.

`TRCDR,hw_out:Trace Point 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 2-26
- ETM_CONFIG on page 2-137
- TRACE on page 2-250
- TRACEBUFFER on page 2-253
- TRACEDATAACCESS on page 2-263
- TRACEDATAWRITE on page 2-273
- TRACEINSTREXEC on page 2-277
- TRACEINSTRFETCH on page 2-281.
2.2.129 TRACEDATAWRITE

The TRACEDATAWRITE command sets a trace point on data reads.

Syntax

```
TRACEDATAWRITE [,qualifier]... [address|address_range]
```

where:

- **qualifier** is an ordered list of zero or more qualifiers. The possible qualifiers are described in Description.
- **address** specifies the address at which the tracepoint is placed.
- **address_range** specifies the address range at which the tracepoint is placed.

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 describing tracing in *RealView Debugger v1.6 Extensions User Guide*.

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.
- **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
  ```


RealView Debugger Commands

**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:

\[
\text{TRCDW, hw_dvalue:0x440 0x1FA00}
\]

**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:

\[
\text{TRCDW, hw_dvalue:0x0, hw_dhigh:0x18 0x1000}
\]

**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 0x400-0x4F0 is written by an instruction at address 0x1000:

\[
\text{TRCDW, hw_dvalue:0x440, hw_dmask:0xF0F 0x1FA00}
\]

**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:2`, or `hw_and:then-2`, where 2 is the tracepoint id of another tracepoint.

In the *and* form, the conditions associated with both tracepoints are tied, 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 tracepoint
- 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:

```plaintext
TRCDW \MODIFY_1\#582
TRCDW,hw_and:then-1 \ACCESS_1\#379
```

**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:

```plaintext
TRCDW,hw_in:"Size of Data Access=Halfword" 0x1E00..0x1FF00
```

**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:

**Trace Point 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:

```plaintext
TRCDW,hw_out:"Trace Point 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.

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

**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 2-26
- **ETM_CONFIG** on page 2-137
- **TRACE** on page 2-250
- **TRACEBUFFER** on page 2-253
- **TRACEDATAACCESS** on page 2-263
- **TRACEDATAREAD** on page 2-268
- **TRACEINSTREXEC** on page 2-277
- **TRACEINSTRFETCH** on page 2-281.
2.2.130 TRACEINSTREXEC

The TRACEINSTREXEC command sets a trace point on instruction execution.

Syntax

\texttt{TRACEINSTREXEC [\textit{, qualifier}]... [\textit{address|address\_range}]}

where:

\textit{qualifier} Is an ordered list of zero or more qualifiers. The possible qualifiers are described in \textit{Description}.

\textit{address} Specifies the address at which the tracepoint is placed.

\textit{address\_range} Specifies the address range at which the tracepoint is placed.

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 \texttt{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 describing tracing in RealView Debugger v1.6 Extensions User Guide.

The command qualifiers are as follows, but not all qualifiers are available for all of the supported trace targets:

\texttt{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.

\texttt{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:

\texttt{TRCIE, hw\_ahigh:0x1200 0x1000}

\texttt{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 an instruction with basic opcode 0x0xEA000040 but with any value in bits [15:8] is executed in code between 0x1FA00-0x1FAFF:

```
TRCIE,hw_dvalue:0x440,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:2`, or `hw_and:then-2`, where 2 is the tracepoint id of another tracepoint.

In the *and* form, the conditions associated with both tracepoints are tied, 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
- `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:

Trace Point 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:

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

**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**

TRCIEEXEC is an alias of TRACEINSTREXEC.

**See also**

The following commands provide similar or related functionality:

- **ANALYZER** on page 2-26
- **ETM_CONFIG** on page 2-137
- **TRACE** on page 2-250
- **TRACEBUFFER** on page 2-253
- **TRACEDATAACCESS** on page 2-263
- **TRACEDATAREAD** on page 2-268
- **TRACEDATAWRITE** on page 2-273
- **TRACEINSTRFETCH** on page 2-281.
2.2.131 TRACEINSTRFETCH

The TRACEINSTRFETCH command sets a trace point on instruction fetch from memory.

Syntax

\[
\text{TRACEINSTRFETCH} \ [,\text{qualifier}] \ [\text{address}\mid\text{address\_range}]
\]

where:

- **qualifier** Is an ordered list of zero or more qualifiers. The possible qualifiers are described in Description.
- **address** Specifies the address at which the tracepoint is placed.
- **address\_range** Specifies the address range at which the tracepoint is placed.

Description

This command sets a tracepoint at the address or address range you specify that triggers when an instruction is 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 \text{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 describing tracing in *RealView Debugger v1.6 Extensions User Guide*.

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.

- **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 an instruction with basic opcode 0x0xEA000040 but with any value in bits 8-15 is fetched in code between 0x1FA00-0x1FAFF:
TRCIF,hw_dvalue:0x440,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:2, or hw_and:then-2, where 2 is the tracepoint id of another tracepoint.

In the and form, the conditions associated with both tracepoints are tied, 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
- 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:

**Trace Point 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:

\[ \text{TRCIF, hw_not:data, hw_dmask:0x00FF} \ldots \]

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

\[ \text{TRCIF, hw_not:addr 0x10040..0x10060} \]

This means to trace execution at addresses other than the range 0x10040 to 0x10060.

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

Alias

TRCIFETCH is an alias of TRACEINSTRFETCH.

See also

The following commands provide similar or related functionality:

- ANALYZER on page 2-26
- ETM_CONFIG on page 2-137
- TRACE on page 2-250
- TRACEBUFFER on page 2-253
- TRACEDATAACCESS on page 2-263
- TRACEDATAREAD on page 2-268
- TRACEDATAWRITE on page 2-273
- TRACEDATAWRITE on page 2-273
- TRACEINSTREXEC on page 2-277.
2.2.132 UNLOAD

The UNLOAD command unloads a specified file.

Syntax

UNLOAD [,all][,symbols_only][,image_only][,killtasks][,nokilltasks] [filename | file_ID] [=task]

where:

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.
killtasks Applicable only to RTOS and when threads identify the entry function. Attempts to identify any threads connected to a file being unloaded and stop their execution.
nokilltasks Does not attempt to identify affected threads and stop their execution.

filename | file_ID

Specifies a file to be unloaded.

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 identifier, 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.

Examples

The following examples show how to use UNLOAD:

unload c:\source\debug\dhry.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 2-21
- LOAD on page 2-168
- RELOAD on page 2-208.
2.2.133 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 will 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 further. When you have moved up one or more levels, you can use the DOWN command (see page 2-115) 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 2-115
- CONTEXT on page 2-89
- DTFILE on page 2-121.
2.2.134 VCLEAR

The VCLEAR command clears a window and sets the cursor to home.

Syntax

VCLEAR windowid

where:

windowid

Specifies the window to be cleared. Valid values for windowid when used in the VCLEAR command are:

50–1024 Window or file number. For further information see VOPEN on page 2-294.

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 2-192
- VCLOSE on page 2-289
- VOPEN on page 2-294
- VSETC on page 2-296.
2.2.135 VCLOSE

The VCLOSE command removes and closes a window or file.

Syntax

VCLOSE window

where:

window Specifies the window to be closed. Valid values for window lie in the range 50–1024.

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 2-149
- PRINTF on page 2-192
- VCLEAR on page 2-288
- VOPEN on page 2-294
- VSETC on page 2-296.
2.2.136 VERIFYFILE

The VERIFYFILE command compares the contents of a specified file with the contents of target memory.

Syntax

VERIFYFILE ,[obj|raw|ascii] [,opts] name [=address/offset]

where:

obj
The file is an executable file in the standard target format. For ARM targets, this is ARM-ELF.
There are no opts supported for this file type.

raw
The file is a stream of 8-bit values that are written to target memory without further interpretation.
There are no opts supported for this file type.

ascii
The file is a stream of ASCII digits separated by whitespace. The interpretation of the digits is specified by other qualifiers. The starting address of the file must be specified in a bracketed line 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.

The following opts are supported for this file type:

byte The file is a stream of 8-bit values that are written to target memory without further 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.

name Specifies the name of the file to be read.
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 and Oak processors have word 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 2-205
- TEST on page 2-246
- WRITEFILE on page 2-301.
2.2.137 VMACRO

The VMACRO command attaches the output of a macro to a window.

Syntax

VMACRO window [,macro_name(args)]

where:

window Specifies the window to be associated with the macro. Valid values for window lie in the range 50–1024.

macro_name Specifies the name and call arguments of the macro that is invoked when the associated window is updated. This happens whenever:

- A special update request is received from the execution environment.
- Execution stops.

Description

The VMACRO command attaches a specified macro to a specified window. The macro is then executed whenever it is called or whenever the window is updated. You can use the FPRINTF command (see page 2-151) to write data to the window.

If you do not supply a macro name, the window 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.

See also

The following commands provide similar or related functionality:

- DEFINE on page 2-97
- FPRINTF on page 2-151
- MONITOR on page 2-181
- PRINTVALUE on page 2-200
- VOPEN on page 2-294
- VSETC on page 2-296.
2.2.138 VOPEN

The VOPEN command creates a window that you can use with commands that have a ;window parameter.

Syntax

VOPEN window [,screen_number,loc_top,loc_left,loc_bottom,loc_right]

where:

window

Specifies a number to identify the new window. If a window already exists with the specified number the command fails.
Integers in the range 50–1024 are valid for identifying the window. Use this value for the ;window parameter in commands that you want to display their output in this window.

screen_number

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_number 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 ;window 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 may 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 the following are true:

- $loc_{top}$ 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 2-149
- `PRINTF` on page 2-151
- `VCLOSE` on page 2-289.
2.2.139 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. Window numbers in the range 50–1024 are available for the window.
- **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 will appear.

**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 2-149
- *PRINTF* on page 2-151
- *VCLOSE* on page 2-289.
2.2.140 WAIT

The WAIT command tells the debugger whether to wait for a command to complete before permitting another command to be issued.

**Syntax**

```plaintext
WAIT = [ON | OFF]
```

where:

- **ON** specifies that all following commands will run synchronously.
- **OFF** specifies that following commands run according to their default behavior.

**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.

**Example**

The following example shows how to use WAIT:

```plaintext
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:

- `DEFINE` on page 2-97.
2.2.141 WARMSTART

WARMSTART is an alias of RESET (see page 2-210).
2.2.142 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\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\Proc_3 Line 355

See also

The following commands provide similar or related functionality:

- CONTEXT on page 2-89
- SCOPE on page 2-216
- SETREG on page 2-223.
2.2.143 WRITEFILE

The WRITEFILE command writes the contents of memory to a file, performing a format conversion if necessary.

Syntax

\[
\text{WRITEFILE \,[OBJ]|\text{raw}|\text{ascii}\ [,\text{opts}] \ [name] = addressrange}
\]

where:

\text{OBJ} \quad \text{Write the file in the standard executable target format. For ARM targets, this is ARM-ELF.}

\text{There are no opts supported for this file type.}

\text{raw} \quad \text{Write the file as a stream of 8-bit values without further interpretation.}

\text{There are no opts supported for this file type.}

\text{ascii} \quad \text{Write the file as a stream of ASCII digits separated by whitespace. The exact format is specified by other qualifiers. The file has a one line header that is compatible with READFILE and VERIFYFILE:}

\text{[start,end,size]} \quad \text{The start address, a comma, the end address, a comma, and a character indicating the size of each value, where } b \text{ is 8 bits, } h \text{ is 16 bits and } l \text{ is 32 bits.}

The following opts are supported for this file type:

\text{byte} \quad \text{The file is a stream of 8-bit hexadecimal values that are written to the file without further interpretation.}

\text{half} \quad \text{The file is a stream of 16-bit hexadecimal values.}

\text{long} \quad \text{The file is a stream of 32-bit hexadecimal values.}

\text{gui} \quad \text{You are prompted to enter the file type with a dialog.}

\text{name} \quad \text{Specifies the name of the file to be written.}

\text{addressrange} \quad \text{The address range in target memory to write to the file.}

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 and Oak processors have word 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:

[0x8000,0x9000,1]
E28F8090 E890000F 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 endianess of the data in the file.

See also

The following commands provide similar or related functionality:

- \textit{FILL} on page 2-144
- \textit{LOAD} on page 2-168
- \textit{SETMEM} on page 2-221
- \textit{READFILE} on page 2-205.
2.2.144 XTRIGGER

The XTRIGGER command controls whether stopping execution of one processor stops execution of other processors.

Syntax

XTRIGGER [,in_disable][,in_enable][,out_disable][,out_enable][,onhost]
[=connections]

where:

in_disable Disable input triggering.
in_enable Enable input triggering.
out_disable Disable output triggering.
out_enable Enable output triggering.
onhost Implement in software. Use this if hardware support is possible and so is used by default, but you require software implementation nevertheless.

connections Identifies one or more connections that this processor can stop or that can cause this processor to stop.

Description

The XTRIGGER command controls cross-triggering of processor start and stop commands. Use it to specify whether stopping execution of one processor stops execution of other processors.

For best performance the processors must exist in a hardware target. If the processors are simulated in software then longer delays are possible between requesting a processor to stop and execution stopping. If you issue the command with no arguments, it displays the state of all connections (in the same way as clicking on the Synch tab in the Connection Control window).

If you issue the command with qualifiers, you have to specify a list of one or more connections to act on. Input triggering means that the processor is stopped by others. Output triggering means that when the processor stops it stops others.

See also

The following commands provide similar or related functionality:

- CONNECT on page 2-83
- SYNCHEXEC on page 2-244.
Chapter 3
Comparison of Commands

This chapter compares the commands supported by the command-line interface of RealView Debugger with those supported by AXD and armsd. It contains the following section:

- RealView Debugger, armsd, and AXD commands on page 3-2.

3.1 RealView Debugger, armsd, and AXD commands

RealView Debugger and AXD are normally driven through graphical user interfaces, but they can also be driven by typed commands. For full details of the commands available in AXD and armsd, see the ADS 1.2 AXD and armsd Debuggers Guide.

Some commands listed here are only similar to each other, and specific features might not be available using the listed command. If you require these features, you are recommended to check the other command descriptions for alternative ways of performing the required task.

The tables in this chapter list the commands available in a specific debugger and the RealView Debugger command that most closely matches it, see:

- Table 3-1 for AXD commands, and their RealView Debugger equivalents
- Table 3-2 on page 3-7 for armsd commands, and their RealView Debugger equivalents.

3.1.1 AXD commands

Look up any AXD command in Table 3-1 for the equivalent RealView Debugger command.

<table>
<thead>
<tr>
<th>AXD commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACKTRACE</td>
<td>WHERE on page 2-299</td>
</tr>
<tr>
<td>BREAK</td>
<td>BREAKEXECUTION on page 2-47</td>
</tr>
<tr>
<td></td>
<td>BREAKINSTRUCTION on page 2-55</td>
</tr>
<tr>
<td></td>
<td>DTBREAK on page 2-119</td>
</tr>
<tr>
<td>CCLASSES</td>
<td>BROWSE on page 2-74</td>
</tr>
<tr>
<td>CFUNCTIONS</td>
<td>BROWSE on page 2-74</td>
</tr>
<tr>
<td>CLASSES</td>
<td>BROWSE on page 2-74</td>
</tr>
<tr>
<td>CLEAR</td>
<td>VCLEAR on page 2-288</td>
</tr>
<tr>
<td>CLEARSTAT</td>
<td>-</td>
</tr>
<tr>
<td>CLEARWATCH</td>
<td>CLEARBREAK on page 2-79</td>
</tr>
<tr>
<td>COMMENT string</td>
<td>FPRINTF on page 2-151</td>
</tr>
</tbody>
</table>

Table 3-1 AXD commands and equivalents
### Table 3-1 AXD commands and equivalents (continued)

<table>
<thead>
<tr>
<th>AXD commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXT</td>
<td>CONTEXT on page 2-89, WHERE on page 2-299</td>
</tr>
<tr>
<td>CONVARIABLES</td>
<td>EXPAND on page 2-140, SCOPE on page 2-216</td>
</tr>
<tr>
<td>CVARIABLES</td>
<td>BROWSE on page 2-74</td>
</tr>
<tr>
<td>DBCINTERNALS</td>
<td>EDITBO on page 2-131</td>
</tr>
<tr>
<td>DISASSEMBLE</td>
<td>DISASSEMBLE on page 2-109, MODE on page 2-180</td>
</tr>
<tr>
<td>ECHO</td>
<td>-</td>
</tr>
<tr>
<td>EXAMINE</td>
<td>DUMP on page 2-126</td>
</tr>
<tr>
<td>FILES</td>
<td>-</td>
</tr>
<tr>
<td>FILLMEM</td>
<td>FILL on page 2-144, SETMEM on page 2-221</td>
</tr>
<tr>
<td>FINDSTRING</td>
<td>SEARCH on page 2-218</td>
</tr>
<tr>
<td>FINDVALUE</td>
<td>SEARCH on page 2-218</td>
</tr>
<tr>
<td>FORMAT</td>
<td>OPTION RADIX on page 2-187, SETTINGS DISASM on page 2-226,</td>
</tr>
<tr>
<td></td>
<td>SETTINGS DSMVALUE on page 2-226</td>
</tr>
<tr>
<td>FUNCTIONS</td>
<td>PRINTSYMBOLES on page 2-195</td>
</tr>
<tr>
<td>GETFILE</td>
<td>READFILE on page 2-205</td>
</tr>
<tr>
<td>GO</td>
<td>GO on page 2-154</td>
</tr>
<tr>
<td>HELP</td>
<td>DCOMMANDS on page 2-94, HELP on page 2-159</td>
</tr>
<tr>
<td>IMAGES</td>
<td>DTFILE on page 2-121</td>
</tr>
<tr>
<td>IMGPROPERTIES</td>
<td>DTFILE on page 2-121</td>
</tr>
<tr>
<td>IMPORTFORMAT</td>
<td>-</td>
</tr>
<tr>
<td>IN</td>
<td>UP on page 2-287</td>
</tr>
</tbody>
</table>
Comparison of Commands

Table 3-1 AXD commands and equivalents (continued)

<table>
<thead>
<tr>
<th>AXD commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET</td>
<td>CEXPRESSION on page 2-77</td>
</tr>
<tr>
<td></td>
<td>SETNAME on page 2-221</td>
</tr>
<tr>
<td>LIST</td>
<td>DISASSEMBLE on page 2-109</td>
</tr>
<tr>
<td></td>
<td>MODE on page 2-180</td>
</tr>
<tr>
<td></td>
<td>DUMP on page 2-126</td>
</tr>
<tr>
<td>LISTFORMAT</td>
<td>-</td>
</tr>
<tr>
<td>LOAD</td>
<td>LOAD on page 2-168</td>
</tr>
<tr>
<td></td>
<td>RELOAD on page 2-208</td>
</tr>
<tr>
<td>LOADBINARY</td>
<td>READMEFILE on page 2-205</td>
</tr>
<tr>
<td>LOADSESSION</td>
<td>READMEFILE on page 2-204</td>
</tr>
<tr>
<td>LOADSYMBOLS</td>
<td>LOAD/NS on page 2-168</td>
</tr>
<tr>
<td>LOWLEVEL</td>
<td>JOURNAL on page 2-163</td>
</tr>
<tr>
<td>LOWLEVEL</td>
<td>PRINTSENT on page 2-195</td>
</tr>
<tr>
<td>MEMORY</td>
<td>DUMP on page 2-126</td>
</tr>
<tr>
<td>OBEY</td>
<td>INCLUDE on page 2-161</td>
</tr>
<tr>
<td>OUT</td>
<td>DOWN on page 2-115</td>
</tr>
<tr>
<td>PARSE</td>
<td>-</td>
</tr>
<tr>
<td>PRINT</td>
<td>PRINTF on page 2-192</td>
</tr>
<tr>
<td></td>
<td>CEXPRESSION on page 2-77</td>
</tr>
<tr>
<td></td>
<td>PRINTVALUE on page 2-200</td>
</tr>
<tr>
<td>PROCESSORS</td>
<td>DTPPROCESS on page 2-123</td>
</tr>
<tr>
<td>PROCPROPERTYS</td>
<td>-</td>
</tr>
<tr>
<td>PUTFILE</td>
<td>WRITEFILE on page 2-301</td>
</tr>
<tr>
<td>QUITDEBGERGER</td>
<td>QUIT on page 2-203</td>
</tr>
<tr>
<td>READSYMS</td>
<td>LOAD/NS on page 2-168</td>
</tr>
<tr>
<td>RECORD</td>
<td>LOG on page 2-171</td>
</tr>
<tr>
<td>REGBANKS</td>
<td>-</td>
</tr>
</tbody>
</table>
### AXD commands and equivalents (continued)

<table>
<thead>
<tr>
<th>AXD commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REGISTERS</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>RELOAD</strong></td>
<td><strong>RELOAD</strong> on page 2-208</td>
</tr>
<tr>
<td><strong>RUN</strong></td>
<td><strong>GO</strong> on page 2-154</td>
</tr>
<tr>
<td><strong>RUNTOPOS</strong></td>
<td><strong>GO</strong> on page 2-154</td>
</tr>
<tr>
<td><strong>SAVEBINARY</strong></td>
<td><strong>WRITEFILE</strong> on page 2-301</td>
</tr>
<tr>
<td><strong>SAVESSESSION</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>SETACI</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>SETBREAKPROPS</strong></td>
<td><strong>BREAKEXECUTION</strong> on page 2-47</td>
</tr>
<tr>
<td></td>
<td><strong>BREAKINSTRUCTION</strong> on page 2-55</td>
</tr>
<tr>
<td><strong>SETIMGPROP</strong> “file” cmdline</td>
<td><strong>ARGUMENTS</strong> on page 2-30</td>
</tr>
<tr>
<td><strong>SETMEM</strong></td>
<td><strong>CEXPRESSSION</strong> on page 2-77</td>
</tr>
<tr>
<td></td>
<td><strong>FILL</strong> on page 2-144</td>
</tr>
<tr>
<td></td>
<td><strong>SETMEM</strong> on page 2-221</td>
</tr>
<tr>
<td><strong>SETPC</strong></td>
<td><strong>SETREG @PC</strong> on page 2-223</td>
</tr>
<tr>
<td><strong>SETPROC</strong></td>
<td><strong>BOARD</strong> on page 2-37</td>
</tr>
<tr>
<td></td>
<td><strong>THREAD</strong> on page 2-248</td>
</tr>
<tr>
<td><strong>SETPROCPROP</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>SETPROCPROP</strong> vector_catch</td>
<td><strong>BGLOBAL</strong> on page 2-33</td>
</tr>
<tr>
<td><strong>SETPROCPROP</strong> semihosting_enabled</td>
<td>-</td>
</tr>
<tr>
<td><strong>SETREG</strong></td>
<td><strong>SETREG</strong> on page 2-223</td>
</tr>
<tr>
<td><strong>SETSOURCEDIR</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>SETWATCH</strong></td>
<td><strong>CEXPRESSSION</strong> on page 2-77</td>
</tr>
<tr>
<td></td>
<td><strong>SETMEM</strong> on page 2-221</td>
</tr>
<tr>
<td><strong>SETWATCHPROPS</strong></td>
<td><strong>BREAKEXECUTION</strong> on page 2-47</td>
</tr>
<tr>
<td></td>
<td><strong>BREAKINSTRUCTION</strong> on page 2-55</td>
</tr>
<tr>
<td><strong>SOURCE</strong></td>
<td><strong>LIST</strong> on page 2-166</td>
</tr>
<tr>
<td><strong>SOURCEDIR</strong></td>
<td>-</td>
</tr>
</tbody>
</table>
### Table 3-1 AXD commands and equivalents (continued)

<table>
<thead>
<tr>
<th>AXD commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACKENTRIES</td>
<td><em>WHERE</em> on page 2-299</td>
</tr>
<tr>
<td>STACKIN</td>
<td><em>UP</em> on page 2-287</td>
</tr>
<tr>
<td>STACKOUT</td>
<td><em>DOWN</em> on page 2-115</td>
</tr>
<tr>
<td>STATISTICS</td>
<td>-</td>
</tr>
<tr>
<td>STEP</td>
<td><em>STEPINSTR</em> on page 2-233</td>
</tr>
<tr>
<td></td>
<td><em>STEPLINE</em> on page 2-235</td>
</tr>
<tr>
<td></td>
<td><em>STEPINSTR</em> on page 2-238</td>
</tr>
<tr>
<td></td>
<td><em>STEP0</em> on page 2-240</td>
</tr>
<tr>
<td>STEPSIZE</td>
<td><em>RUN</em> on page 2-215</td>
</tr>
<tr>
<td>STOP</td>
<td><em>STOP</em> on page 2-243</td>
</tr>
<tr>
<td>TYPE</td>
<td><em>LIST</em> on page 2-166</td>
</tr>
<tr>
<td>UNBREAK</td>
<td><em>CLEARBREAK</em> on page 2-79</td>
</tr>
<tr>
<td>UNWATCH</td>
<td><em>CLEARBREAK</em> on page 2-79</td>
</tr>
<tr>
<td>VARIABLES</td>
<td><em>PRINTTYPE</em> on page 2-198</td>
</tr>
<tr>
<td>WATCHPT</td>
<td><em>BREAKACCESS</em> on page 2-40</td>
</tr>
<tr>
<td></td>
<td><em>BREAKREAD</em> on page 2-60</td>
</tr>
<tr>
<td></td>
<td><em>BREAWRITE</em> on page 2-67</td>
</tr>
<tr>
<td>WHERE</td>
<td><em>CONTEXT</em> on page 2-89</td>
</tr>
<tr>
<td></td>
<td><em>WHERE</em> on page 2-299</td>
</tr>
</tbody>
</table>
3.1.2 armsd commands

Look up any armsd command in Table 3-2 for the equivalent RealView Debugger command.

<table>
<thead>
<tr>
<th>armsd commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>-</td>
</tr>
<tr>
<td>$cmdline = &quot;string&quot;</td>
<td>ARGUMENTS on page 2-30</td>
</tr>
<tr>
<td>$semihosting_enabled = [0</td>
<td>1]</td>
</tr>
<tr>
<td>$vector_catch = [0</td>
<td>1]</td>
</tr>
<tr>
<td>@regname = value</td>
<td>SETREG on page 2-223</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIAS</td>
<td>ALIAS on page 2-23</td>
</tr>
<tr>
<td>ARGUMENTS</td>
<td>EXPAND on page 2-140</td>
</tr>
<tr>
<td>BACKTRACE</td>
<td>WHERE on page 2-299</td>
</tr>
<tr>
<td>BREAK</td>
<td>BREAKEXECUTION on page 2-47</td>
</tr>
<tr>
<td></td>
<td>BREAKINSTRUCTION on page 2-55</td>
</tr>
<tr>
<td>CALL</td>
<td>function_name(params) as an expression.</td>
</tr>
<tr>
<td>COMMENT</td>
<td>FPRINTF on page 2-151</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>SCOPE on page 2-216</td>
</tr>
<tr>
<td>Control-C</td>
<td>STOP on page 2-243</td>
</tr>
<tr>
<td>COPROC</td>
<td>-</td>
</tr>
<tr>
<td>CREGDEF</td>
<td>Edit board file</td>
</tr>
<tr>
<td>CREGISTERS</td>
<td>@regname as an expression.</td>
</tr>
<tr>
<td>CWRITE</td>
<td>SETREG on page 2-223</td>
</tr>
<tr>
<td>EXAMINE</td>
<td>DUMP on page 2-126</td>
</tr>
<tr>
<td>FIND</td>
<td>SEARCH on page 2-218</td>
</tr>
<tr>
<td>FPREGISTERS</td>
<td>@regname for each register as an expression</td>
</tr>
<tr>
<td>GETFILE</td>
<td>READFILE on page 2-205</td>
</tr>
</tbody>
</table>
Comparison of Commands

Table 3-2 armsd commands and equivalents (continued)

<table>
<thead>
<tr>
<th>armsd commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO</td>
<td>GO on page 2-154</td>
</tr>
<tr>
<td>HELP</td>
<td>DCOMMANDS on page 2-94</td>
</tr>
<tr>
<td></td>
<td>HELP on page 2-159</td>
</tr>
<tr>
<td>IN</td>
<td>DOWN on page 2-115</td>
</tr>
<tr>
<td>ISTEP</td>
<td>STEPINSTR on page 2-233</td>
</tr>
<tr>
<td></td>
<td>STEPOINSTR on page 2-238</td>
</tr>
<tr>
<td>LANGUAGE</td>
<td></td>
</tr>
<tr>
<td>LET</td>
<td>SETREG on page 2-223</td>
</tr>
<tr>
<td></td>
<td>CEXPRESSION on page 2-77</td>
</tr>
<tr>
<td>LIST</td>
<td>DISASSEMBLE on page 2-109</td>
</tr>
<tr>
<td></td>
<td>MODE on page 2-180</td>
</tr>
<tr>
<td></td>
<td>DUMP on page 2-126</td>
</tr>
<tr>
<td>LOAD</td>
<td>LOAD on page 2-168</td>
</tr>
<tr>
<td>LOG</td>
<td>LOG on page 2-171</td>
</tr>
<tr>
<td>LSYM</td>
<td>PRINTSYMBOLS on page 2-195</td>
</tr>
<tr>
<td>lvalue_memoryaddress = data_value</td>
<td>CEXPRESSION on page 2-77</td>
</tr>
<tr>
<td>OBEY</td>
<td>INCLUDE on page 2-161</td>
</tr>
<tr>
<td>OUT</td>
<td>UP on page 2-287</td>
</tr>
<tr>
<td>PAUSE</td>
<td>PAUSE on page 2-191</td>
</tr>
<tr>
<td>PRINT</td>
<td>CEXPRESSION on page 2-77</td>
</tr>
<tr>
<td></td>
<td>PRINTF on page 2-192</td>
</tr>
<tr>
<td></td>
<td>PRINTVALUE on page 2-200</td>
</tr>
<tr>
<td>PROFCLEAR</td>
<td>-</td>
</tr>
<tr>
<td>PROFOFF</td>
<td>-</td>
</tr>
<tr>
<td>PROFON</td>
<td>-</td>
</tr>
<tr>
<td>PROFWRITE</td>
<td>-</td>
</tr>
<tr>
<td>PUTFILE</td>
<td>WRITEFILE on page 2-301</td>
</tr>
</tbody>
</table>
### Table 3-2 armsd commands and equivalents (continued)

<table>
<thead>
<tr>
<th>armsd commands</th>
<th>RealView Debugger commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>QUIT</td>
<td>QUIT on page 2-203</td>
</tr>
<tr>
<td>READSYMS</td>
<td>LOAD/NS on page 2-168</td>
</tr>
<tr>
<td>REGISTERS</td>
<td><code>@regname</code> for each register as an expression</td>
</tr>
<tr>
<td>RELOAD</td>
<td>RELOAD on page 2-208</td>
</tr>
<tr>
<td>RETURN</td>
<td>-</td>
</tr>
<tr>
<td>STEP</td>
<td>STEPLINE on page 2-235</td>
</tr>
<tr>
<td></td>
<td>STEPO on page 2-240</td>
</tr>
<tr>
<td>SYMBOLS</td>
<td>EXPAND on page 2-140</td>
</tr>
<tr>
<td></td>
<td>PRINTSYMBOLES on page 2-195</td>
</tr>
<tr>
<td>TYPE</td>
<td>LIST on page 2-166</td>
</tr>
<tr>
<td>UNBREAK</td>
<td>CLEARBREAK on page 2-79</td>
</tr>
<tr>
<td>UNWATCH</td>
<td>CLEARBREAK on page 2-79</td>
</tr>
<tr>
<td>VARIABLE</td>
<td>PRINTTYPE on page 2-198</td>
</tr>
<tr>
<td>WATCH</td>
<td>BREAKACCESS on page 2-40</td>
</tr>
<tr>
<td></td>
<td>BREAKREAD on page 2-60</td>
</tr>
<tr>
<td></td>
<td>BREAKWRITE on page 2-67</td>
</tr>
<tr>
<td>WHERE</td>
<td>CONTEXT on page 2-89</td>
</tr>
<tr>
<td>WHILE</td>
<td>WHILE statement (in macro or include file).</td>
</tr>
</tbody>
</table>
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.

Angel
Angel is a software debug monitor that runs on the target and enables you to debug applications running on ARM-based hardware. Angel is commonly used where a JTAG emulator, such as Multi-ICE, is not available.

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.
ARM state  A processor that is executing ARM (32-bit) instructions is operating in ARM state.

See also Thumb state

ARMulator  ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.

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.

ATPCS  ARM-Thumb Procedure Call Standard.

Backtracing  See Stack Traceback.

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.

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.

CPSR  Current Program Status Register.

See also Program Status Register.
Debug With Arbitrary Record Format (DWARF)
ARM code generation tools generate debug information in DWARF2 format.

Deprecated
A deprecated option or feature is one that you are strongly discouraged from using. Deprecated options and features will not be supported 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.

**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.

**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 focused on silicon chip testing methods. Many debug and programming tools use a *Joint Test Action Group (JTAG)* interface port to communicate with processors. For further information refer to 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.

**Multi-ICE**
A JTAG-based tool for debugging embedded systems.

**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.

RDI
See Remote Debug Interface.

RealView Compilation Tools
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
A software product add-on to RealView Debugger that extends the debugging capability with the addition of real-time program and data tracing.

Remote Debug Interface (RDI)
The Remote Debug Interface is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:
- a simulator running on the host (for example, ARMulator)
- a debug monitor running on ARM-based hardware accessed through a communication link (for example, Angel)
- a debug agent controlling an ARM processor through hardware debug support (for example, Multi-ICE).

Remote_A
Remote_A is a software protocol converter and configuration interface. It converts between the RDI 1.5 software interface of a debugger and the Angel Debug Protocol used by Angel targets. It can communicate over a serial or Ethernet interface.

RTOS
Real Time Operating System.

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.
### Glossary

**Simulator**
A simulator executes non-native instructions in software (simulating a core).

**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 allows 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 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 state
A processor that is executing Thumb (16-bit) instructions is operating in Thumb state.
See also ARM state

TLA
See Tektronix Logic Analyzer.

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 need to capture data at the same bandwidth as that of an analyzer monitoring the core buses directly.

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.

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.

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.
**Glossary**

**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.
The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A
Abbreviations of commands 1-10
ADD 1-23
ADD command 1-23, 2-17
ADDBOARD command 2-20
Adding symbols to debugger symbol table 2-17
ADDFILE command 2-21, 2-103
Address of nonlabel symbol 1-18
ALIAS command 2-23
Analysis window time units 2-256
ANALYZER command 2-26
ARGUMENTS command 2-30
ARM Developer Suite vii
ARM ETM 2-26, 2-251
AXD scripts 1-3

B
BACCESS alias 1-26, 2-32
Batched commands 1-7
BEXECUTION alias 2-32
BGLOBAL command 2-33
BINSTRUCTION alias 2-36
BOARD command 2-37
Board files and configuration 2-2
BREAD alias 2-39
BREAK alias 2-39
BREAKACCESS command 2-40
BREAKEXECUTION command 2-47
BREAKINSTRUCTION command 2-55
Breakpoint triggering a macro 2-40, 2-47
Breakpoints
data abort 2-34
illegal access 2-34
memory access 2-40
software interrupt 2-34
Breakpoints, address-range 2-49
Breakpoints, deleting 2-79
Breakpoints, disabling 2-107
Breakpoint, setting 2-47
BREAKREAD command 2-60
BREAKWRITE command 2-67
BROWSE command 2-74
BWRITE alias 2-75

C
C language program flow keywords 1-11
Calling target functions 1-19
Command line using 1-6
CEXPRESSION command 2-77
Changing the current board 2-37
CLEARBREAK command 2-79
CLI commands 2-8
Clock speed, processor 2-256
Command abbreviations 1-10
alphabetical list 2-12
parameters 1-9
qualifiers 1-9
syntax 1-6
Index

Command aliases  1-11
Command history  1-6
Command line
  debugger options on  1-7
  location  1-3
Command pane  1-3
Command priority  2-24
Command quoting  1-10
Command syntax  1-8
Commands flags, qualifiers, and parameters  1-9
Comma, in qualifiers  1-9
Comments from users  x
COMPARE command  2-81
Configuration and board files  2-2
CONNECT command  1-5, 2-83
Connect to target  2-83
CONTEXT command  1-24, 2-89
Conventions, typographical  vi
Coprocessor tracing  2-138
COPY command  2-91
Creating files and text writing  2-9
Current connection  2-37
Custom window  2-51, 2-57
Custom windows  2-44
Cycle-accurate tracing  2-138

D
Data-type conversions  1-21
DBOARD alias  2-93
DBREAK alias  2-93
DCOMMANDS  1-4
DCOMMANDS command  2-94
Debugger expressions  1-13, 1-14
Debugger macro  1-16
Debugger variables  1-16
DEFINE command  2-97
DELBOARD command  2-100
DELETE command  2-17, 2-101
Deleting breakpoints  2-79
DELFILE command  2-103
Dereferencing variables  1-26
DHELP alias  1-4, 2-106
DISABLEBREAK command  2-107, 2-109, 2-134
Disabling breakpoints  2-107
DISASSEMBLE command  1-27, 2-109
DISCONNECT command  2-111
DLOADERR command  2-113
DMAP command  2-114
DOWN command  2-115
DIPIPEVIEW command  2-117
DTBOARD command  2-118
DTBREAK command  2-119
DTFILE command  2-121
DTPROCESS command  2-123
DTRACE command  2-125
DUMP command  2-126, 2-128, 2-131, 2-132
DUMPMAP command  2-128
DVFILE alias  2-130
DVPROCESS alias  2-130

E
EDITBOARDFILE command  2-131
EmbeddedICE  2-42, 2-43, 2-49, 2-50, 2-61, 2-62
EMURESET command  2-132
EMURST alias  2-133
ENABLEBREAK command  2-134
Entering commands directly  1-6
ERROR command  2-136, 2-143
ETM_CONFIG command  2-137
Examining source files  2-4
Executable image file list  2-21, 2-103, 2-208, 2-285
Execution control  2-3
Execution tracepoint  2-263
EXPAND command  1-27, 2-140
Expressions  1-14
Extended Target Visibility  1-8

F
FAILINC command  2-143
Feedback  x
FIFO high-water mark  2-137
FILL command  2-144
Filling memory  2-144
Filtering the Analysis window  2-258
FLASH command  2-147
Flash Control dialog  2-147
Flash memory, writing  2-147
FOPEN command  2-149
Format of syntax definition  2-12
PRINTF command  1-24, 2-151

G
Glossary  Glossary-1
GO command  1-27, 2-154
GOSTEP command  2-156
GUI
  command-line equivalent  1-3

H
HALT command  2-158
HELP command  2-159
Help on Commands  1-4
Help, online  1-4
HWRESET alias  2-160

I
INCLUDE command  1-8, 2-97, 2-161
Include files  1-7
Indirect addresses  1-18
Instruction address breakpoints  2-47
Interactive command entry  1-6
INTRPT command  2-76
Invoking a batch file  1-7

J
JOURNAL command  2-163
JUMP command  2-165

K
Keywords  1-11
Index

L
Language data types 1-21
Linker map file 2-128
LIST command 1-27, 2-166
List of commands 2-12
Little-endian memory 2-145
LOAD command 1-23, 2-168
LOAD commands 2-21
Location of CLI 1-3
LOG command 2-171

ONSTATE command 2-185
OPTION command 2-187, 2-188

P
Parameters of commands 1-9
PATHTRANSLATE command 2-190
PAUSE command 2-191
Physical to Logical Address Mapping 2-26
PRINTF command 2-192
PRINTSYMBOLS command 1-23, 1-24, 2-195
PRINTTYPE command 2-198, 2-204
PRINTVALUE command 1-22, 1-24, 1-27, 2-200, 2-203
Process information 2-123
Processor context 2-89
Processor events 2-33
Processor execution pipe-line 2-117
Processor tracing commands 2-10
Program image management 2-5
Program symbol manipulation 2-8
Program symbols 1-15
PROPERTIES alias 2-202
PS alias 2-202
PT alias 2-202

Q
Qualifiers
use of comma 1-9
use of slash 1-9
Qualifiers of commands 1-9
QUIT command 2-203
Quoting symbols 1-14

R
READBOARDFILE command 2-100, 2-204
READFILE command 2-205, 2-206
REEEXEC alias 2-207
References to symbols 1-15
Referencing stack variables 1-27
Referencing variables 1-25
RELOAD command 2-208
RELOAD commands 2-21
RESET command 2-210
RESETBREAKS command 2-211
Resource Viewer window 2-45
RESTART command 2-213
ROM-based breakpoints 2-47
RSTBREAKS alias 2-214
RTOS loader 2-21
RUN command 2-215

S
Scope 2-89
SCOPE command 2-216
Scope of symbols 1-21
Scoping rules 1-21
Scripts 1-7
SEARCH command 2-218
Searching for trigger 2-256
Select Analysis Configuration 2-26
Semihosting 2-30
Set execution breakpoint 2-47
SETFLAGS command 2-220
SETMEM command 2-221
SETREG command 2-223
SETTINGS command 2-226
Set/Edit Event Triggers 2-26
SHOW command 2-230
SIMULATOR command 2-231
Single-step the program 2-156
SINSTR alias 2-232
Size of Data Access 2-265
Slash, in qualifiers 1-9
SM alias 2-232
SOINSTR alias 2-232
SOVERLINE alias 2-232
Special operators 1-25
SR alias 2-232
START command 2-158, 2-243
Status enquiry commands 2-7
STEPINSTR command 2-233, 2-235, 2-238, 2-240, 2-245
STEPLINE command 2-235
STEPO command 2-240
STEPOINSTR command 2-238
STOP command 2-243
SWI exceptions 2-34

Oak processors 2-34
Online help 1-4

ARM DUI 0175C Copyright © 2002, 2003 ARM Limited. All rights reserved.
Symbol length, max   1-14
Symbol names   1-14
Symbol storage class   1-20
Symbols   1-14
Symbols, quoting   1-14
Symbols, underscore   1-15
SYNCH command   2-244
Syntax definition, format   2-12
Syntax of commands   1-6
Syntax, of CLI   1-8

T
Target functions
   calling   1-19
Target processor clock speed   2-256
Target registers and memory access commands   2-6
TeakLite processors   2-34
Terminology  Glossary-1
TEST command   2-246
THREAD command   2-248
Trace analyzer   2-253
Trace analyzer information   2-125
TRACE command   2-250
Trace Point Type   2-265, 2-283
Trace records, set number of   2-255
TRACEBUFFER command   2-253
TRACEDATAACCESS command  2-263
TRACEDATAREAD command  2-268
TRACEDATAWRITE command  2-273
TRACEINSTREXEC command  2-277
TRACEINSTRFETCH command  2-281
Trigger, finding   2-256
Trigger, trace   2-26
Type casting   1-22
Types of expressions   1-13
Typographical conventions  vi

UP command   2-287
User’s comments   x
Using the CLI   1-2

V
VCLEAR command   2-288, 2-289
VCLOSE command   2-289
VERIFYFILE command   2-290, 2-291
VMACRO command   2-292
VOPEN command   2-294
VSETC command   2-296

W
WAIT command   2-297
WARMSTART alias   2-298
WHERE alias   2-299
Window specification   1-10
Write flash blocks   2-147
WRITEFILE command   2-301

X
XTRIGGER command   2-303

U
Underscore in source symbols   1-15
Units for Analysis window time   2-256
UNLOAD command   2-285