# ARM® Developer Suite

**Version 1.2**

## AXD and armsd Debuggers Guide

ARM®

Copyright © 1999-2001 ARM Limited. All rights reserved.
ARM DUI 0066D

# ARM Developer Suite
## AXD and armsd Debuggers Guide

Copyright © 1999-2001 ARM Limited. All rights reserved.

### Release Information

The following changes have been made to this document.

**Change History**

| Date | Issue | Change |
|---|---|---|
| October 1999 | ARM DUI 0066A | ADS 1.0 Release |
| March 2000 | ARM DUI 0066B | ADS 1.0.1 Release |
| November 2000 | ARM DUI 0066C | ADS 1.1 Release |
| November 2001 | ARM DUI 0066D | ADS 1.2 Release |

### 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.

# Contents
# ARM Developer Suite AXD and armsd Debuggers Guide

**Preface**
About this book ..... viii
Feedback ..... xii

**Part A AXD**

**Chapter 1 About AXD**
1.1 Debugger concepts ..... 1-2
1.2 Interfacing with targets ..... 1-5
1.3 Debugging systems ..... 1-8
1.4 Availability and compatibility ..... 1-11
1.5 Online help ..... 1-12

**Chapter 2 Getting Started in AXD**
2.1 License-managed software ..... 2-2
2.2 Starting and closing AXD ..... 2-3
2.3 Debugger target ..... 2-6
2.4 AXD displays ..... 2-9
2.5 AXD menus ..... 2-12

 *Copyright © 1999-2001 ARM Limited. All rights reserved.* 

2.6 Tool icons, status bar, keys, and commands ..... 2-14

**Chapter 3 Working with AXD**

3.1 Running a demonstration program ..... 3-2
3.2 Setting a breakpoint ..... 3-4
3.3 Setting a watchpoint ..... 3-6
3.4 Examining the contents of variables ..... 3-8
3.5 Examining the contents of registers ..... 3-12
3.6 Examining the contents of memory ..... 3-14
3.7 Locating and changing values and verifying changes ..... 3-16
3.8 Creating a revised version of the program ..... 3-18

**Chapter 4 AXD Facilities**

4.1 Stopping and stepping ..... 4-2
4.2 Expressions ..... 4-4
4.3 Viewing and editing ..... 4-6
4.4 Entering addresses ..... 4-12
4.5 Persistence ..... 4-13
4.6 RealMonitor support ..... 4-14
4.7 Data formatting ..... 4-16
4.8 Profiling ..... 4-27

**Chapter 5 AXD Desktop**

5.1 Menus, toolbars, and status bar ..... 5-2
5.2 File menu ..... 5-6
5.3 Search menu ..... 5-16
5.4 Processor Views menu ..... 5-18
5.5 System Views menu ..... 5-48
5.6 Execute menu ..... 5-76
5.7 Options menu ..... 5-80
5.8 Window menu ..... 5-99
5.9 Help menu ..... 5-102

**Chapter 6 AXD Command-line Interface**

6.1 Command Line Window ..... 6-2
6.2 Parameters and prefixes ..... 6-4
6.3 Commands with list support ..... 6-5
6.4 Predefined command parameters ..... 6-6
6.5 Definitions ..... 6-9
6.6 Commands ..... 6-13

# Part B armsd

**Chapter 7 About armsd**

7.1 About armsd ..... 7-2

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

7.2 Command syntax ..... 7-3

**Chapter 8 Getting Started in armsd**
8.1 Specifying source-level objects ..... 8-2
8.2 armsd variables ..... 8-7
8.3 Low-level debugging ..... 8-13

**Chapter 9 Working with armsd**
9.1 Groups of armsd commands ..... 9-2
9.2 Alphabetical list of armsd commands ..... 9-7
9.3 Accessing the debug communications channel ..... 9-46
9.4 armsd commands for EmbeddedICE ..... 9-47

**Appendix A AXD and armsd Commands**
A.1 Comparison of commands ..... A-2
A.2 Useful internal variables ..... A-8

**Appendix B Coprocessor Registers**
B.1 ARM710T processor ..... B-2
B.2 ARM720T processor ..... B-3
B.3 ARM740T processor ..... B-4
B.4 ARM920T Rev 0 processor ..... B-5
B.5 ARM920T Rev 1 processor ..... B-7
B.6 ARM940T Rev 0 processor ..... B-9
B.7 ARM940T Rev 1 processor ..... B-11
B.8 ARM946E-S processor ..... B-13
B.9 ARM966E-S processor ..... B-15
B.10 ARM10200E processor ..... B-16
B.11 ARM1020E processor ..... B-20
B.12 ARM10E processor ..... B-22
B.13 XScale processor ..... B-24

**Appendix C Supplementary Display Module Formats**
C.1 Predefined formats ..... C-2
C.2 User-defined formats ..... C-5

**Appendix D Using the Flash Downloader**
D.1 About the Flash downloader ..... D-2
D.2 Using the Flash downloader from AXD ..... D-4
D.3 Using the Flash downloader from armsd ..... D-5
D.4 Setting the IP address of a PID board ..... D-6

**Glossary**

 *Copyright © 1999-2001 ARM Limited. All rights reserved.* 

Copyright © 1999-2001 ARM Limited. All rights reserved.

# Preface

This preface introduces the ARM debuggers and their documentation. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page xii.


*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## About this book

This book has two parts that describe the currently supported ARM debuggers:

- Part A describes the graphical user interface components of *ARM eXtended Debugger* (AXD). This is the most recent ARM debugger and is part of the *ARM Developer Suite* (ADS). Tutorial information is included to demonstrate the main features of AXD. If AXD is the only debugger you use, you can safely ignore Part B, but you might have to refer to the Appendixes, Glossary, and Index at the end of the book.
- Part B describes the *ARM Symbolic Debugger* (armsd).

### Intended audience

This book is written for developers who are using either of the currently supported ARM debuggers under MS Windows NT, 95, 98, 2000, or UNIX. It assumes that you are an experienced software developer, and that you are familiar with the ARM development tools as described in *Getting Started* (see *ARM publications* on page x).

### Using this book

This book is organized into the following parts and chapters:

**PART A** Part A covers the use of AXD.

**Chapter 1** ***About AXD***

Chapter 1 explains some of the concepts of debugging and the terminology used. It also describes the ARM debuggers, AXD and armsd, and how this book is complemented by online help.

**Chapter 2** ***Getting Started in AXD***

Chapter 2 reminds you that you use ARM software under a license agreement, and how software licensing is managed. It then explains how to set up a debugger target, and gives an overview of the AXD desktop.

**Chapter 3** ***Working with AXD***

Chapter 3 provides some examples with step-by-step instructions to demonstrate typical debugging sessions.

**Chapter 4** ***AXD Facilities***

Chapter 4 starts with an overview of the debugging facilities that you must have, and how they are provided by AXD. This is followed by information about expressions, viewing and editing data, and profiling.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

**Chapter 5** ***AXD Desktop***

Chapter 5 describes the menus, views, dialogs, and tool and status bars provided by the AXD desktop.

**Chapter 6** ***AXD Command-line Interface***

Chapter 6 describes command-line operation of AXD.

**PART B** Part B covers the use of armsd.

**Chapter 7** ***About armsd***

Chapter 11 introduces armsd. This is an interactive, command-line, debugger that provides source-level debugging for C and C++, and low-level support for ARM assembly language.

**Chapter 8** ***Getting Started in armsd***

Chapter 12 explains how to set up and start using armsd, and describes some necessary debugging concepts.

**Chapter 9** ***Working with armsd***

Chapter 13 provides detailed descriptions of the features of armsd, and instructions for their use.

**Appendix A** ***AXD and armsd Commands***

Appendix A lists and compares all the commands available in both the armsd and AXD debugger.

**Appendix B** ***Coprocessor Registers***

Appendix B describes the various available coprocessor registers.

**Appendix C** ***Supplementary Display Module Formats***

Appendix C describes how supplementary display formats can be defined in files that can be read by AXD.

**Appendix D** ***Using the Flash Downloader***

Appendix D describes how the Flash downloader can be used to write a binary file to the Flash memory on an ARM Integrator™ board, or an ARM Development (PID) board.

**Glossary** An alphabetically arranged glossary defines the special terms used.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## Typographical conventions

The following typographical conventions are used in this book:

| | |
|---|---|
| *italic* | Highlights important notes, introduces special terminology, denotes internal cross-references, and citations. |
| **bold** | Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate. |
| `monospace` | Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code. |
| `monospace` | Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name. |
| *`monospace italic`* | Denotes arguments to commands and functions where the argument is to be replaced by a specific value. |
| **`monospace bold`** | Denotes language keywords when used outside example code. |

## Further reading

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

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current information.

See also the ARM Frequently Asked Questions list at `http://www.arm.com`.

### ARM publications

This book contains information specific to the ARM debuggers supplied with ADS. The ADS document suite describes other components of ADS in the following books:

- *ADS Installation and License Management Guide* (ARM DUI 0139)
- *Getting Started* (ARM DUI 0064)
- *ADS Assembler Guide* (ARM DUI 0068)
- *ADS Compilers and Libraries Guide* (ARM DUI 0067)
- *ADS Linker and Utilities Guide* (ARM DUI 0151)


Copyright © 1999-2001 ARM Limited. All rights reserved.

- *CodeWarrior IDE Guide* (ARM DUI 0065)
- *ADS Debug Target Guide* (ARM DUI 0058)
- *ADS Developer Guide* (ARM DUI 0056)
- *ARM Applications Library Programmer's Guide* (ARM DUI 0081).

The following additional documentation is provided with the ARM Developer Suite:

- *ARM Architecture Reference Manual* (ARM DDI 0100). This is supplied both in DynaText format, and in PDF format in `install_directory\PDF\DDI0100E_ARM_ARM.pdf`
- *ARM ELF specification* (SWS ESPC 0003). This is supplied in PDF format in `install_directory\PDF\specs\ARMELF.pdf`.
- *TIS DWARF 2 specification*. This is supplied in PDF format in `install_directory\PDF\specs\TIS-DWARF2.pdf`.
- *ARM/Thumb® Procedure Call Standard (ATPCS) Specification (SWS ESPC 0002)*. This is supplied in PDF format in `install_directory\PDF\specs\ATPCS.pdf`.

In addition, refer to the following for specific information relating to ARM products:

- *ARM Reference Peripherals Specification* (ARM DDI 0062)
- *ARM Trace Debug Tools User Guide* (ARM DUI 0118)
- ARM Agilent Debug Interface Version 1.0 User Guide
- the ARM datasheet or technical reference manual for your hardware device.

### Third party products

Further information on Agilent emulators and similar products is available from Agilent at `http://www.agilent.com`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

# Feedback

ARM Limited welcomes feedback on both the ARM Developer Suite, and its documentation.

## Feedback on the ARM Developer Suite

If you have any problems with the ARM Developer Suite, please contact your supplier. To help us provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type, and version
- a small stand-alone sample of code that reproduces the problem
- 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
- the version string of the tool, including the version number and date.

## Feedback on this book

If you have any problems with this book, please 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 also welcome.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

# Part A

## AXD

# Chapter 1
# About AXD

This chapter explains some of the concepts of debugging and the terminology used. It also describes the two ARM debuggers, and how this book is complemented by online help. It contains the following sections:


- *Debugger concepts* on page 1-2
- *Interfacing with targets* on page 1-5
- *Online help* on page 1-12.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 1.1 Debugger concepts

This section introduces some of the concepts involved in debugging program images.

### 1.1.1 Debugger

A debugger is software that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. This part of the book covers AXD, the *ARM eXtended Debugger*. The second part of this book covers armsd, the *ARM Symbolic Debugger*.

### 1.1.2 Debug target

At an early stage of product development there might be no hardware. The expected behavior of the product is simulated by software. Even though you might run this software on the same computer as the debugger, it is useful to think of the target as a separate piece of hardware.

Alternatively, you can build a prototype product on a printed circuit board, including one or more processors on which you run and debug software.

You build the finished product only when you are satisfied with the performance, proved by hardware or software simulation.

The debugger issues instructions that can:

- load software into memory on the target
- start and stop execution of that software
- display the contents of memory, registers, and variables
- enable you to change stored values.

The form of the target is immaterial to the debugger as long as the target obeys these instructions in exactly the same way as the final product.

### 1.1.3 Debug agent

A debug agent performs the actions requested by the debugger, for example:

- setting breakpoints
- reading from memory
- writing to memory.

The debug agent is *not* the program being debugged, or the debugger itself.

Examples of debug agents include:

- Multi-ICE®

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

- ARMulator™
- Angel™.

Multi-ICE is a separate product. It is not supplied with ADS.

### 1.1.4 Remote debug interface

The *Remote Debug Interface* (RDI) is an ARM standard procedural interface between a debugger and the debug agent (see Figure 1-1 on page 1-6).

RDI gives the debugger a uniform way to communicate with:

- a debug agent 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).

For more details on using this interface please contact ARM at `http://www.arm.com`.

### 1.1.5 Single-processor hardware

In many cases, the target has only a single processor. All ARM debuggers can operate successfully on single-processor targets.

### 1.1.6 Multi-processor hardware

There is a growing requirement for multi-processor hardware:

- certain processors might be dedicated to particular tasks
- parallel processing might be appropriate and beneficial.

In these cases the debugger must allow you to examine and control the processes happening simultaneously in a number of processors.

### 1.1.7 Contexts

Each processor in the target can have a process currently in execution. Each process uses values stored in variables, registers, and other memory locations. These values can change during the execution of the process.

The *context* of a process describes its current state, as defined principally by the call stack that lists all the currently active calls. When a function is called, and again when control is returned, the context changes.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Because variables can have class, local, or global scope, the context determines which variables are currently accessible.

Every process has its own context. When execution of a process stops, you can examine and change values in its current context.

### 1.1.8 Scope

The scope of a variable is determined by the point within a program at which it is defined. Variables can have values that are relevant within:

- a specific class only (*class*)
- a specific function only (*local*)
- a specific file only (*static global*)
- the entire process (*global*).


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 1.2 Interfacing with targets

AXD enables you to run and debug your ARM-targeted image using any of the debugging systems described in *Debugging systems* on page 1-8.

Refer to the documentation supplied with your target board for specific information on setting up your system to work with the ARM Developer Suite, and Multi-ICE, Angel, and so on.

Most of this part of the book applies to both the Windows and the UNIX version of AXD. The term AXD refers to either version. If a section applies to one version only, this is indicated in the text or in the section heading.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 1.2.1 Debugging an ARM application

AXD works in conjunction with either a hardware or a software target system, as shown in Figure 1-1.

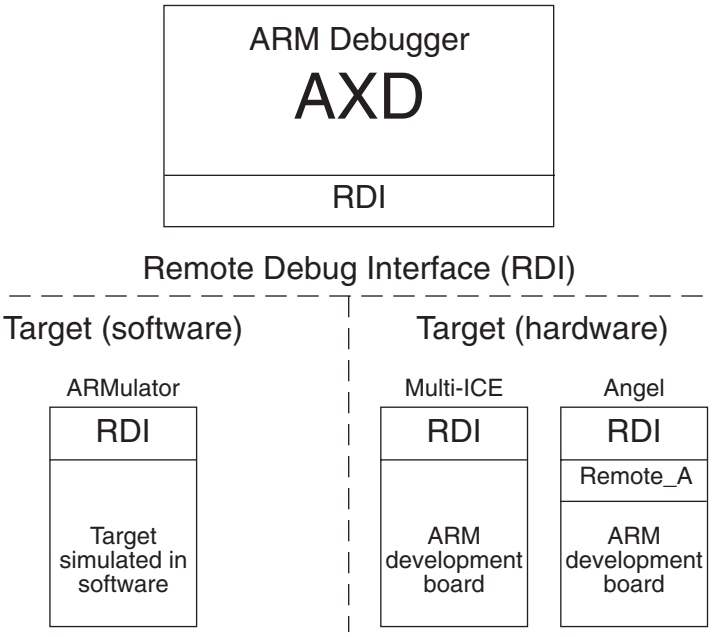



**Figure 1-1 Debugger-target interface**

An ARM development board, communicating through Multi-ICE, or Angel, is an example of a hardware target system. ARMulator is an example of a software target system.

You debug your application using a number of windows giving you various views on the application you are debugging.

To debug your application you must choose:

- a *debugging system*, that can be either:
  - hardware-based on an ARM core
  - software that simulates an ARM core.
- a *debugger,* such as AXD, or armsd.

Figure 1-2 on page 1-7 shows a typical debugging arrangement of hardware and software.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

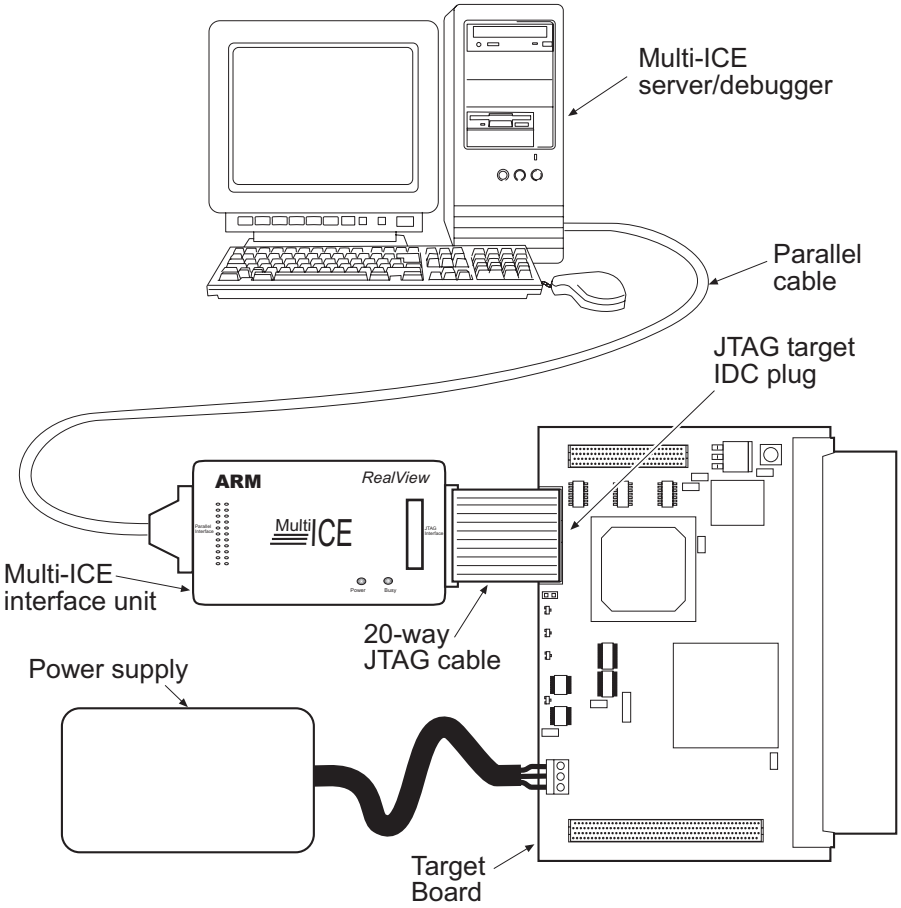



**Figure 1-2 A typical debugging set-up**


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 1.3 Debugging systems

The following are debugging systems for applications developed to run on ARM cores:

- *ARMulator*
- *Multi-ICE® and EmbeddedICE®*
- *Angel debug monitor* on page 1-9
- *ARM Agilent Debug Interface* on page 1-9.

See *Configure Target...* on page 5-87 for information about the configuration of debugger target systems.

### 1.3.1 ARMulator

ARMulator is a collection of programs that simulate the instruction sets and architecture of various ARM processors. ARMulator:

- provides an environment for the development of ARM-targeted software on the supported host systems
- enables benchmarking of ARM-targeted software.

ARMulator is *instruction-accurate*, meaning that it models the instruction set without regard to the precise timing characteristics of the processor. It can report the number of cycles the hardware would take. See the *ADS Debug Target Guide* for more information.

### 1.3.2 Multi-ICE® and EmbeddedICE®

Multi-ICE and EmbeddedICE are JTAG-based debugging systems for ARM processors. They provide the interface between a debugger and an ARM core embedded within an *Application Specific Integrated Circuit* (ASIC).

**Note**

The EmbeddedICE product is no longer sold. It has been replaced by Multi-ICE.

These systems provide:

- real-time address-dependent and data-dependent breakpoints
- single stepping
- full access to, and control of the ARM core
- full access to the ASIC system
- full memory access (read and write)
- full I/O system access (read and write).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Multi-ICE can debug applications running in either ARM state or Thumb state on target hardware. Refer to Multi-ICE documentation for detailed information.

Multi-ICE and EmbeddedICE also enable the embedded microprocessor to access services of the host system, such as screen display, keyboard input, and disk drive storage, using semihosting.

### 1.3.3 Angel debug monitor

Angel is a debug monitor that allows rapid development and debugging of applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state on target hardware. It runs alongside the application being debugged on the target platform.

Angel also enables the embedded microprocessor to access services of the host system, such as screen display, keyboard input, and disk drive storage, using semihosting.

You can use Angel to debug an application on an ARM Development Board or on your own custom hardware. See the *ADS Debug Target Guide* for more information.

### 1.3.4 ARM Agilent Debug Interface

ARM Agilent Debug Interface (ARM ADI) can be purchased as an extension to ADS and enables AXD to communicate with an Agilent emulation probe or emulation module for debugging applications running on ARM cores.

The Agilent emulation probe is a standalone emulator whereas the emulation module is installed as part of an Agilent logic analyzer such as one of the 16700 series. However, both probe and module connect to a JTAG debug port on the target system through a *Target Interface Module* (TIM).

The emulator provides a variety of debug facilities such as run control and access to both memory and to CPU and coprocessor registers. AXD accesses these facilities across an Ethernet connection. For further information on Agilent emulators and similar products see *Third party products* on page xi.

**Note**

For the ARM7 and ARM9 cores, you must use the Agilent E3459A emulation probe (or 16610A emulation module) and Agilent E3459-66501 TIM.

For technical information and support of the emulator, contact Agilent or one of its authorized agents.


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Note**

ARM Agilent Debug Interface is a separate product. It is not supplied with ADS.

ARM Agilent Debug Interface handles all aspects of setting up and maintaining the Ethernet connection with the emulator. For further information on configuring this connection, refer to the documentation accompanying the product, for example, the *ARM Agilent Debug Interface Version 1.0 User Guide*.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 1.4 Availability and compatibility

ARM products undergo continual development and improvement, and two debuggers are currently available and fully supported.

The ADS CD-ROM includes the ARM debuggers:

- AXD (both Windows and UNIX versions)
- armsd (ARM Symbolic Debugger).

AXD is the recommended debugger. It provides additional functionality that is not available in armsd.

The main improvements in AXD are:

- a completely redesigned graphical user interface offering multiple views
- a new command-line interface.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 1.5 Online help

Online help complements the information contained in this guide.

Information about the ARM debuggers appears in this book and online with the following differences:

- this book concentrates on overall concepts, tutorial material, and descriptions of facilities
- online help complements the information provided in this book, and provides finer details relating to such topics as individual data entry fields, check boxes, and buttons.

When you are running AXD, use online help to obtain information about your current situation. You can also navigate your way to any other pages of available online help.

### 1.5.1 Displaying online help

You can display online help in any of the following ways:

**F1 key** Press the F1 key on your keyboard to display online help on the currently active window.

**Help button** Many windows contain a **Help** button that you can click to display help relevant to that window.

**Help menu** The **Help** menu is shown in Figure 1-3.

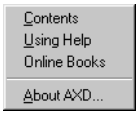



**Figure 1-3 Help menu**

Select **Contents** to display the first page of AXD online help. You can navigate from there to any available topic.

Select **Using Help** to display a guide to the use of online help.

Select **Online Books** to start running browser software that enables you to display online copies of the printed manuals that you received with AXD. This is equivalent to selecting **Start → Programs → ARM Developer Suite v1.2 → Online Books**.

Select **About AXD...** to display details of the version of AXD that you are running.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

If you are licensed to use the *Trace Debug Tools* (TDT) and your target processor supports trace, the option **TDT Help** is also shown on the online **Help** menu.

**Query tools**

Click on the **Query** tool in the **Help** toolbar as an alternative to selecting **Contents** from the **Help** menu.

Click on the **Query and arrow** tool in the **Help** toolbar to change the mouse pointer into a query and arrow, then click again on any item on the screen for which you want help.

**Dialog help**

Dialogs within AXD include a **Query** tool that you can use to display field level help. Clicking on this button changes the mouse pointer into a **Query and arrow** pointer which you can then click on any item for which you want help. You can also display a help pop-up box by placing the mouse pointer over an item and pressing F1.

**Hypertext links**

Most pages of online help include highlighted text that you click on to display related online help:

- highlighted plain text displays a pop-up box
- highlighted underscored text causes a jump to another page of help.

**Related topics button**

Many pages of online help include a **Related topics** button that you can click to display a new window containing links to related online help.

**Browse buttons**

Most pages of online help include a pair of browse buttons allowing you to display a sequence of related help pages.


Copyright © 1999-2001 ARM Limited. All rights reserved.

Copyright © 1999-2001 ARM Limited. All rights reserved.

# Chapter 2
# Getting Started in AXD

This chapter describes how to start running AXD, set up your debugger target, and operate the AXD desktop. It contains the following sections:


- *License-managed software* on page 2-2
- *Starting and closing AXD* on page 2-3
- *Debugger target* on page 2-6
- *AXD displays* on page 2-9
- *AXD menus* on page 2-12
- *Tool icons, status bar, keys, and commands* on page 2-14.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 2.1 License-managed software

Some software is locked, preventing you from running it, until you have been granted a license to use it. If you require a license you can obtain it quickly by applying for it by email.

You can use some license-managed software with a temporary license which places a time limit on your use of the software.

Details of license-managed software, how licensing works, and how to apply for a license are explained in the *ADS Installation and License Management Guide*.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## 2.2 Starting and closing AXD

This section describes:

- *Starting AXD*
- *AXD arguments*
- *Closing AXD* on page 2-5.

### 2.2.1 Starting AXD

Start AXD in any of the following ways:

- If you are running Windows, double-click on the **AXD Debugger** icon or select **Start → Programs → ARM Developer Suite v1.2 → AXD Debugger**.
- If you are working in the CodeWarrior IDE, refer to the *CodeWarrior IDE Guide* for more information on starting AXD.
- If you are running under UNIX, launch AXD from a shell, optionally with arguments (see *AXD arguments*). Either:
  - from any directory type the full path and name of the debugger, for example, `/opt/arm/axd`
  - change to the directory containing the debugger and type its name, for example, `./axd`
- launch AXD from MS-DOS or from a Command Prompt window, optionally with arguments (see *AXD arguments*)
- create a shortcut, optionally with arguments.

### 2.2.2 AXD arguments

The syntax for the command-line method of starting AXD is as follows (any arguments must be in lowercase):

```
axd [-logo|-nologo] [-session session_file_name]
[-debug|-exec|-script script_name] [-halt|-nohalt|-attach] [-restore_default]
[-clear_registry] [-help] [image_name [parm1 [parm2 [...]]]]
```

where:

`-logo` Displays the splash screen and is the default setting.

`-nologo` Suppresses the display of the splash screen.

`-session` *`session_file_name`*

Specifies a file in which an earlier debug session was saved. You must give the full pathname to the required file and, if the filename includes spaces, you must enclose it in quotes. The earlier session is restored to the state it was in when it was saved. Any images in the session file are loaded, and connection to the target is established.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

`-debug` Loads the image and sets a breakpoint on `main()`. It does not take an argument. Use this after one or more of the following:

- default session loaded
- session explicitly loaded from command line using `-session`
- image explicitly loaded from command line.

`-exec` Starts execution of the loaded image with the entry point. It does not take an argument. Use this after one or more of the following:

- default session loaded
- session explicitly loaded from command line using `-session`
- image explicitly loaded from command line.

`-script` *`script_name`*

Obeys the commands in file *`script_name`*. This is the equivalent of typing `obey` *`script_name`* in the CLI system view as soon as the debugger starts up. Use this after one or more of the following:

- default session loaded
- session explicitly loaded from command line using `-session`
- image explicitly loaded from command line.

`-halt` Connects AXD to the target and stops execution of the target.

`-nohalt` Connects AXD to the target without stopping execution of the target. This is possible only with targets that support RealMonitor. If connection of AXD might stop execution of the target, then the attempt to connect is abandoned.

`-attach` Connects AXD to the target. Execution of the target is not stopped if the target supports RealMonitor. Execution of other targets stops when the connection is made.

`-restore_default`

Starts AXD without reference to the default session file. AXD starts with default windows displayed in a default layout.

`-clear_registry`

Starts AXD without reference to the default session file. AXD starts with default windows displayed in a default layout. In addition, all existing target configuration information is deleted.

`-help` Displays text describing how to use the AXD command.

*`image_name`* Specifies a file containing an image to be loaded. You must place this name, followed by any required parameters, at the end of the command, because the remainder of the command passes to the image, not to AXD.

*`parm1, parm2, ...`*

Any parameters required by *`image_name`*.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

#### Examples

To restore a debug session saved in file `friday.ses`, type:

```
axd -session friday.ses
```

To launch AXD and load `img01.axf` with arguments 10, 3.14159, and ABC, type:

```
axd img01.axf 10 3.14159 'ABC'
```

### 2.2.3 Closing AXD

Close down AXD in any of the following ways:

- Select **Exit** from the **File** menu.
- Click the **X** button at the far right of the AXD title bar (not available in UNIX).
- Press ALT-F4.
- Double-click on the icon in the top left corner of the main window.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 2.3 Debugger target

This section explains how to set up the target hardware, or simulator, which runs the software to be debugged, using:

- *ARMulator*
- *Multi-ICE unit and target board* on page 2-7
- *Angel or EmbeddedICE* on page 2-7.

The first time you run AXD, ARMulator is selected by default as the target, with default settings taken from a configuration file (`install_directory\bin\ARMulate.cnf`). Subsequently, AXD starts up with the last used target configuration by default. You can modify this behavior by starting AXD with arguments (see *AXD arguments* on page 2-3).

**Note**

In some of these procedures you use a browse dialog to locate and select a required file such as `armulate.dll`. Some files, including DLLs, are not listed by default. Select **Windows Explorer → View → Options... → Show all files** to display system files.

### 2.3.1 ARMulator

If you install ADS and run AXD, an ARMulator debugging session starts by default, with ARMulator configured by settings held in a default configuration file.

To reconfigure ARMulator, or to return to ARMulator after using another debug agent:

1. Select **Configure Target...** from the AXD **Options** menu. You are prompted to choose a target, in a dialog similar to that shown in Figure 2-1.

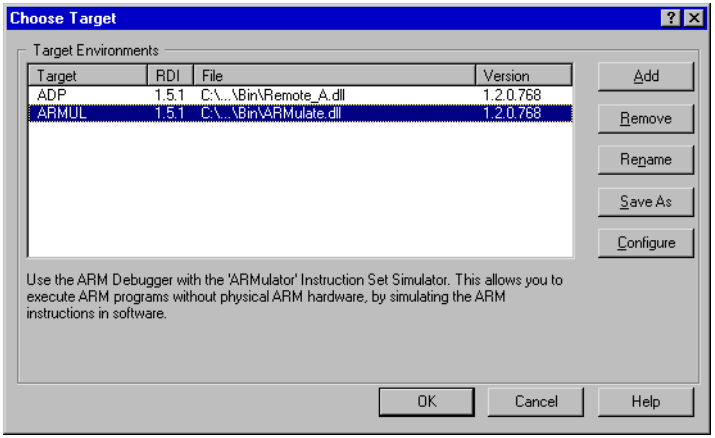



**Figure 2-1 Selecting a target**


Copyright © 1999-2001 ARM Limited. All rights reserved.

2. Select the ARMUL target. If ARMUL is not in the list of available target environments, click **Add**, locate and select `armulate.dll`, click **Open**, and ARMUL is added to the list and selected.
3. Click the **Configure** button to examine or change the ARMulator configuration settings. The resulting dialog is described in *Configure Target...* on page 5-87.
4. Click **OK** when you have selected ARMUL as the target, and configured it if necessary. You can now load an image onto the target and control its execution.

### 2.3.2 Multi-ICE unit and target board

To set up a hardware target of this kind for the first time, refer to the *ARM Multi-ICE User Guide*. When the hardware is correctly connected and configured, start a debugging session as follows:

1. Connect Multi-ICE to your target board with the JTAG connector. Switch on the power supply to your target board (for example, an ARM Integrator board). Multi-ICE is usually configured to get its power from the target board.
2. Run the Multi-ICE server software on the computer that has the Multi-ICE hardware unit connected to its parallel port.
3. Select **Auto-configure** from the **File** menu, and check that the software detects the processors that you expect to find on the target board.
4. Select **Configure Target...** from the **Options** menu. You are prompted to choose a target, in a dialog similar to that shown in Figure 2-1 on page 2-6.
5. In the **Choose Target** dialog select **Multi-ICE**. If Multi-ICE is not yet in the list of available target environments, click **Add**, locate and select `Multi-ICE.dll`, click **Open**, and Multi-ICE is added to the list and selected.
6. If this is the first time you have used this target, or the target configuration has changed since your last debugging session, click the **Configure** button. The resulting dialog is described in *Configure Target...* on page 5-87.
7. When you have selected Multi-ICE as the target, and configured it if necessary, click **OK**. You can now load an image onto the target and control its execution.

The debugger internal variable `$top_of_memory` has a default value of 0x80000. This is the required value when you are using an ARM Development (PID) board as the target. An ARM Integrator target requires `$top_of_memory` to have a value of 0x40000. Other targets might require different values. To change the value of `$top_of_memory`, see *Debugger Internals system view* on page 5-69.

### 2.3.3 Angel or EmbeddedICE

To start an Angel or EmbeddedICE debugging session:

1. Ensure your target board (for example, an ARM Integrator board) is correctly configured and connected to your computer, then switch on its power supply.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

2. Select **Configure Target...** from the **Options** menu in AXD. You are prompted to choose a target, in a dialog similar to that shown in Figure 2-1 on page 2-6.
3. Select the *Angel Debug Protocol* (ADP) target. If ADP is not yet in the list of available target environments, click **Add**, locate and select `remote_a.dll` click **Open**, and ADP is added to the list and selected.
4. Click the **Configure** button if this is the first time you have used this target, or the target configuration has changed since your last debugging session. The resulting dialog is described in *Configure Target...* on page 5-87.
5. Click **OK** when you have selected ADP as the target, and configured it if necessary. You can now load an image onto the target and control its execution.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 2.4 AXD displays

This section describes the various kinds of displays that you see when using AXD:

- *Views*
- *Viewing structured data*
- *Multiple Document Interface* on page 2-10
- *Docked and floating windows* on page 2-10
- *Tabbed pages* on page 2-10
- *Dialogs* on page 2-11.

### 2.4.1 Views

Various *views* allow you to examine and control the processes you are debugging.

In the main menu bar, two menus contain items that display views:

- The items in the **Processor Views** menu display views that apply to the current processor only, and are described in *Processor Views menu* on page 5-18.
- The items in the **System Views** menu display views that apply to the entire, possibly multiprocessor, target system and are described in *System Views menu* on page 5-48.

### 2.4.2 Viewing structured data

In Registers, Variables, and Watch views, you often see data displayed in a tree structure that you can expand or collapse. Generally, values that have changed since the previous break in execution are colored red, but that is not possible in the following situations:

- You might collapse a branch of displayed data in a Registers view, continue execution for one or more steps, and then expand the branch again. In this case the values displayed in red are those that have changed since the last time they were displayed, not since the previous break in execution. Also any value that changed and returned to its original value is not colored red.
- You might collapse a branch of displayed data in a Variables or Watch view, continue execution for one or more steps, and then expand the branch again. The old values are discarded if execution takes place with a collapsed branch, and recalculated when you later expand the display. In this case, therefore, it is impossible to know which values have changed, so no red coloring is possible.

If you try to expand a branch that has no elements, the string `(Empty)` is displayed.


Copyright © 1999-2001 ARM Limited. All rights reserved.

The expansion dialog imposes limits. Any single expansion is restricted to a maximum of 4000 elements, and you normally request far fewer than that. The total number of elements supported in a single view is 32,000.

### 2.4.3 Multiple Document Interface

AXD uses the Windows *Multiple Document Interface* (MDI) so that you can display several windows at the same time. This enables you to view a wide range of information at the same time, such as registers, variables, and execution context. You can arrange the debugger windows in different ways so that, for example, some are docked, some are free-floating, and the remainder are cascaded or tiled.

### 2.4.4 Docked and floating windows

Source and disassembly views appear as floating windows, but most views that you display appear first as docked windows. Right-click anywhere within a window to display its pop-up menu. The pop-up menu of every view that you can dock has an **Allow docking** item. This is initially checked showing that it is selected.

A docked window is attached to one edge of the main window, with a width and height dependent upon any other docked windows that are sharing the same screen edge.

If you click the **Allow docking** item of the pop-up menu so that it is unchecked, the window floats. Another pop-up menu item, **Float within main window**, allows you to specify whether a floating window is restricted to the main window or can float anywhere on the screen.

Windows that are floating within the main window are the only ones that you can reposition and resize by selecting **Cascade** or **Tile** from the **Window** menu.

### 2.4.5 Tabbed pages

Several AXD dialogs and property sheets use tabbed pages. These allow displays that contain a large number of data entry fields, control buttons, check boxes, and radio buttons to be presented in parts.

Although you view only one page at a time, the tabs of all the pages are visible. Click on any tab to bring its page to the front of the display. You can switch between tabbed pages as often as you like while making settings or entering data.

You can consider all the tabbed pages in a display to be parts of a single large display.

Any changes you make become effective only when you click the **OK** button or the **Apply** button. Click the **Cancel** button (or its equivalent) to abandon any changes made on all tabbed pages in the display.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 2.4.6 Dialogs

AXD uses dialogs frequently. A dialog is a convenient way of grouping together a number of fields, lists, check boxes, and buttons, allowing you to make changes to several related fields or values at the same time.

When you select a menu item that operates in this way, a suitable dialog appears. Enter values, select from lists, select and deselect check boxes until you are satisfied with all the settings. The new settings become effective only when you click the **OK** button or the **Apply** button. You can click the **Cancel** button (or its equivalent) to abandon any changes you have made and leave all settings unchanged. The dialog disappears automatically when you finish using it.

The AXD dialogs are shown and described in Chapter 5 *AXD Desktop*.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 2.5 AXD menus

To invoke the main features of AXD, you select menu items in one of the following ways:

- use the mouse to pull down a menu from the main menu bar near the top of the screen and highlight the required item, then click to select the item
- press the Alt key, use the arrow keys to select the required menu and highlight the required item, then press the Return or Enter key to select the item
- hold down the Alt key while you press the key of the underlined character in the required menu name, then press the key of the underlined character of the required item to select it.

Other menus are the pop-up menus associated with each view, as described in *Pop-up menus* on page 2-13.

### 2.5.1 Menu bar menus

The menus available from the menu bar are:

**File** Enables you to transfer data between the target system and disk files, or to exit from AXD.

**Search** Enables you to search for a specified character string, either in the memory of a process or in a specified disk file.

**Processor Views**
Enables you to select a view to open on the currently selected processor.

**System Views**
Enables you to select a system-wide view to open.

**Execute** Enables you to control execution of a program image, or to set or toggle watchpoints, or to toggle or delete all breakpoints.

**Options** Enables you to:

- set the disassembly mode
- configure the debugger user interface, target system, and processor properties
- maintain a list of directories that are searched to find source files
- enable or disable the display of the status bar
- enable or disable the collection of profiling information.

**Window** Enables you to control how MDI windows and icons are displayed, and to set refresh options.

**Help** Enables you to display online help on the use of AXD, or identify the version of AXD that you are running.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Each of these main menus is described in detail in Chapter 5 *AXD Desktop*.

### 2.5.2 Pop-up menus

In addition to the menus listed in the main menu bar, each view has one or more pop-up context menus offering additional items.

You generally display pop-up menus by right-clicking anywhere within a view. However, the pop-up menu items that are enabled can depend on the window item currently selected, if any, or on the position of the mouse pointer when you right-click.

Each pop-up menu is described and shown in Chapter 5 *AXD Desktop* as part of the description of each view. Online help gives further information.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 2.6 Tool icons, status bar, keys, and commands

This section introduces:

- *Toolbars*
- *Tooltips*
- *Status bar*
- *Keyboard shortcuts*
- *In-place editing* on page 2-15
- *Command-line interface* on page 2-16.

### 2.6.1 Toolbars

Most of the main menus have corresponding toolbars with icons representing most of their items. To choose which menus are duplicated as toolbars, or to hide toolbars:

1. Select **Configure Interface** from the **Options** menu.
2. Click the check boxes under **Toolbars** so that the toolbars you want are checked.
3. Click the **OK** button.

To alter the order in which the toolbars are displayed, or reposition them on the screen, place the mouse pointer in a toolbar but not on an icon, then drag it to its new position.

When a toolbar is docked at one of the edges of the screen, it is only one icon high (or wide), but when it is floating and you change its shape, its icons automatically regroup.

### 2.6.2 Tooltips

If you leave the mouse pointer positioned on a toolbar icon for a few seconds without clicking, a tooltip appears informing you of the purpose of the icon. In addition, in disassembly and source views, you can leave the mouse pointer positioned over a variable or register to display the value of the variable or register as a tooltip.

### 2.6.3 Status bar

The status bar is a single line in which AXD can display several items of relevant information at the bottom of the debugger screen when appropriate (see *Status bar contents* on page 5-5).

You can display or hide the status bar (see *Status Bar display control* on page 5-98).

### 2.6.4 Keyboard shortcuts

Several kinds of keyboard shortcuts are described in *AXD menus* on page 2-12.


Copyright © 1999-2001 ARM Limited. All rights reserved.

In addition, most items in three main menus (**Processor Views**, **System Views**, and **Execute**), and many items in pop-up menus, also show keys or key combinations that allow you to select that item directly, without first pulling down the menu. For example, pressing:

- Ctrl+R displays a Registers processor view
- Alt+O displays an Output system view
- F9 toggles a breakpoint on or off.

You can expand list views with the + and - keys. Look at the menus to see all the available keyboard shortcuts.

### 2.6.5 In-place editing

In-place editing allows you to see most clearly what you are doing when you change a stored value. It is used whenever possible. For example, when you are displaying the contents of memory or registers, and want to change a stored value:

1. Double-click on the value you want to change, or press Enter if the item is already selected. The value is enclosed in a box with the characters highlighted to show they are selected.
2. Either enter data to overwrite the highlighted data, or press the left or right arrow keys to deselect the existing data and position the insertion point where you want to amend the existing data.
3. Press Enter or Return to store the new value in the selected location.

If you press Escape or move the focus elsewhere instead of pressing Enter or Return, then any changes you made in the highlighted field are ignored.

In-place editing is not appropriate for:

- editing complex data where some prompting is helpful
- editing groups of related items
- selecting values from predefined lists.

In these cases an appropriate dialog is displayed.

See *Data formatting* on page 4-16 for details on editing data formats.

#### In-place editing under UNIX

AXD can appear to hang when using in-place editing under Solaris. This might happen if the focus is changed, for example, if you double-click on a register to change its contents and then double-click on a different register.

To correct this, you should modify the X-Windows configuration file located in your `$HOME` directory. The file needs to contain the following line:

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

```
Dtwm*secondariesOnTop: True
```

The case is important so enter the line exactly as shown. If the `.Xdefaults` file does not exist then you should create it.

### 2.6.6 Command-line interface

The *Command Line Interface* (CLI) window is an alternative to the graphical user interface. In the CLI window you can:

- enter commands in response to prompts
- view data that you have requested
- submit a file in which you have set up a sequence of commands.

See Chapter 6 *AXD Command-line Interface* for details.


Copyright © 1999-2001 ARM Limited. All rights reserved.

# Chapter 3
# Working with AXD

This chapter gives step-by-step instructions to perform a variety of debugging tasks. You might find it useful to follow all the instructions, as a tutorial. Chapter 5 *AXD Desktop* gives further details of specific features.

This chapter contains the following sections:


- *Running a demonstration program* on page 3-2
- *Setting a breakpoint* on page 3-4
- *Setting a watchpoint* on page 3-6
- *Examining the contents of variables* on page 3-8
- *Examining the contents of registers* on page 3-12
- *Examining the contents of memory* on page 3-14
- *Locating and changing values and verifying changes* on page 3-16
- *Creating a revised version of the program* on page 3-18.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 3.1 Running a demonstration program

Various demonstration projects are supplied, with programs in the form of ARM assembly language, C, or C++ source code files. These projects are stored in subdirectories of `Examples` in the `ADSv1_2` installation directory.

The examples given in this chapter have all been tested and shown to work as described. Your hardware and software might not be the same as those used for testing these examples, so it is possible that certain addresses or values might vary slightly from those shown, and some of the examples might not apply to you. In these cases you might need to modify the instructions to suit your own circumstances.

You are likely to be using software such as ARMulator to simulate a debugger target. Alternatively, your target might consist of a Multi-ICE unit and an ARM Integrator board. If so, you should have set up the hardware and the software as described in *Multi-ICE unit and target board* on page 2-7. In all cases, you must have selected the target you intend to use and configured it, as described in *Configure Target...* on page 5-87.

The following instructions show you how to build, load, and execute a demonstration program that runs the Dhrystone test software:

1. Create an executable image by compiling the source code files in the `Dhry` subdirectory and linking the resulting objects with the libraries that they use. If you are running under Windows you can use the CodeWarrior IDE project file `dhry.mcp` supplied. This organizes your work into projects and largely automates the tasks of creating and maintaining different versions of a program.

2. Run AXD, by selecting **Debug** from the **Project** menu of the CodeWarrior IDE if that is how you built the image file `dhry.axf`. This invokes the AXD debugger with the image loaded.

   Alternatively, run AXD separately, select **Load Image...** from the **File** menu to display the Load Image dialog, navigate to the directory of the `dhry.axf` image file, select the file, and click **Open**. The image loads into memory on the target, so the selected processor can execute it.

   A Disassembly processor view of the image is displayed as shown in Figure 3-1 on page 3-3.

   A blue arrow indicates the current execution point.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

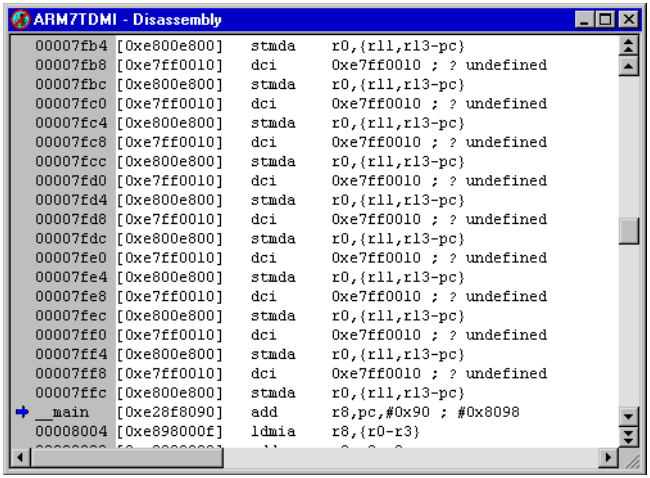



**Figure 3-1 AXD with Disassembly processor view**

3. Select **Go** from the **Execute** menu (or press F5) to begin execution on the target processor. Execution stops at the beginning of function `main()`, where a breakpoint is set by default. A red disc and a marker indicate the line where a breakpoint is set.

4. Also, a Source processor view of the relevant few lines of the relevant file is displayed. If it is not, right-click in the Disassembly view and select **Source** from the pop-up menu. Again, a red disc and marker indicate the line where a breakpoint is set, and a blue arrow indicates the current execution point.

5. Select **Go** from the **Execute** menu (or press F5) again to continue execution. You are prompted, in the Console processor view, for the number of runs through the benchmark that you want performed. Enter 8000. The program runs for a few seconds, displays some diagnostic messages, and shows the test results.

6. To repeat the execution of the program, select **Reload Current Image** from the **File** menu, then repeat Steps 3, 4, and 5. You do not have to open the Source process view again. Once opened, it remains displayed.

   If you are running AXD on a very fast machine, you might have to increase the number of runs through the benchmark (to 25000 or more, perhaps), to make the process last long enough to time accurately.

For details of the program, refer to the `readme.txt` file and the various source files in the `Dhry` subdirectory.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 3.2 Setting a breakpoint

This example runs the same program again, this time with a breakpoint that stops execution a few times. You can examine values when execution stops.

1. Select **Reload Current Image** from the **File** menu.

2. Select **Go** from the **Execute** menu (or press F5) to reach the first breakpoint, set by default at the beginning of function `main()` and indicated by a red disc. You can see the source file `dhry_1.c` with a breakpoint and the current position indicated by a red marker at line number 78.

3. Scroll down through the source file until line number 150 is visible. This is a call to `Proc_4()`, and is inside the loop to be executed the number of times you specify.

4. Right-click on line 150 to position the cursor there and display the pop-up menu, and select **Toggle Breakpoint** (or left-click on the line and press F9, or double-click in the margin next to the line). Another red disc and marker indicate that you have set a second breakpoint, as shown in Figure 3-2.

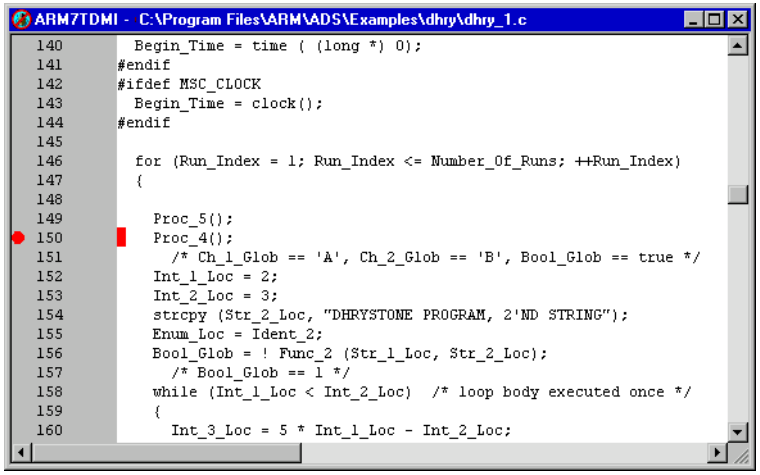



**Figure 3-2 Breakpoint set inside loop**

5. Select **Breakpoints** from the **System Views** menu to edit the details of the new breakpoint. The breakpoints pane is displayed.

   Double-click on the line in the breakpoints pane that describes the new breakpoint, or right-click on it and select **Properties**, to display a Breakpoint Properties dialog.

   Enter `750` in the out of... field in the Condition group, as shown in Figure 3-3 on page 3-5. This is the number of times execution has to arrive at the breakpoint to trigger it.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Click **OK**.

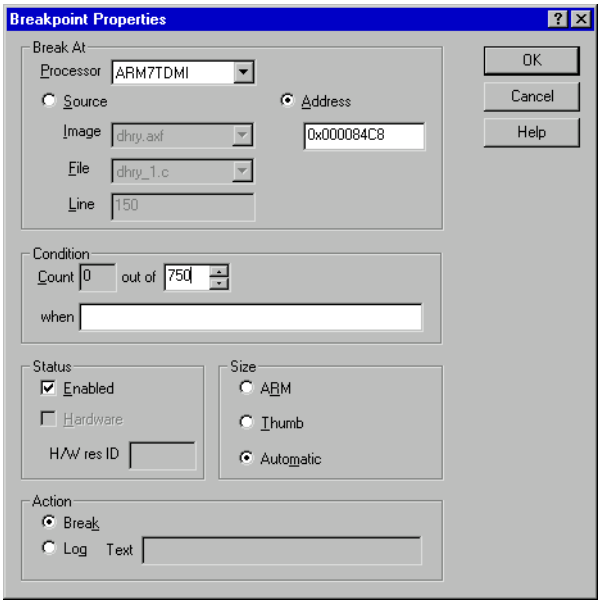



**Figure 3-3 Setting breakpoint details**

6. Press F5 to resume execution, and enter the smaller number of 5000 this time for the number of runs required. Execution stops the 750th time your new breakpoint is reached.

7. Select **Variables** from the **Processor Views** menu to check progress. Reposition or resize the window if necessary. Click the **Local** tab and look for the `Run_Index` variable. Its value is shown as `2EE` (hexadecimal). Right-click on the variable so that it is selected and a pop-up menu appears. Select **Format** → **Decimal** and the value is now displayed as 750 (decimal).

8. Press F5 to resume execution, and the value of the `Run_Index` local variable changes to 1500. It is now colored to show that its value has changed since the previous display.

9. Press F5 repeatedly until the value of `Run_Index` reaches the highest multiple of 750 before exceeding your specified number of runs, then once more to allow the program to complete execution. (This time the Dhrystone test results are meaningless, because of the interruptions to the timing measurements, but the use of a breakpoint has been demonstrated.)

10. Close down the Breakpoints system view, either by right-clicking and selecting **Close** or by clicking on the **Close** button in the title bar if the view is not docked.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 3.3 Setting a watchpoint

This example runs the same program again, this time with a watchpoint that stops execution a few times. You can examine values when execution stops.

1. Select **Reload Current Image** from the **File** menu.
2. Select **Go** from the **Execute** menu (or press F5) to reach the first breakpoint, set by default at the beginning of function `main()` and indicated by a red disc and marker. You can see the source file `dhry_1.c` with a breakpoint and the current position indicated at line number 78.
3. Select **Go** from the **Execute** menu (or press F5) to continue execution.
4. Enter `770` when you are prompted for the number of runs to execute. Execution continues until it reaches the breakpoint at line 150 for the 750th time. This is the breakpoint you defined in *Setting a breakpoint* on page 3-4.
5. Select **Watchpoints** from the **System Views** menu, right-click in the Watchpoints system view, and select **Add** to display the Watchpoint Properties dialog (see Figure 3-4). For this example you specify that execution stops every sixth time the value of `Run_Index` changes.

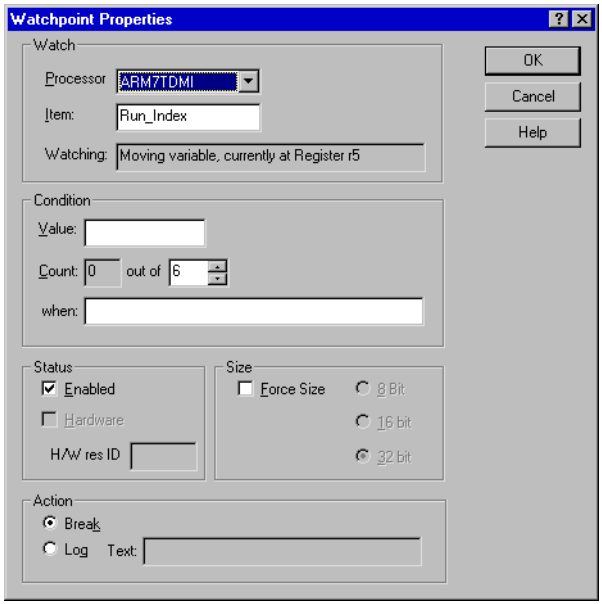



**Figure 3-4 Setting a watchpoint**

Enter `Run_Index` in the Item field in the Watch group.


Copyright © 1999-2001 ARM Limited. All rights reserved.

Set the out of... field in the Condition group to a value of `6`. This is the number of times the watched value has to change to trigger the watchpoint action.

Click the **OK** button.

6. Select **Variables** from the **Processor Views** menu if the Variables processor view is not already displayed. Reposition or resize the window if necessary. Click the **Local** tab and look for the `Run_Index` variable.

   The value of `Run_Index` is currently `750`. If it is displayed in hexadecimal notation, right-click on the value and select **Format** → **Decimal** to change the display format to decimal.

7. Press F5 to resume execution. Soon the value of the `Run_Index` local variable changes to `756`. It is now displayed in red to show that its value has changed since the previous display. Execution stops.

8. Examine any displayed values, then press F5 again to resume execution and perform six more runs. When the value of `Run_Index` becomes greater than the number of runs you specified, the test results are displayed and execution terminates. (Again, the Dhrystone test results are meaningless, because of the interruptions to the timing measurements, but the use of a watchpoint has been demonstrated.)

9. Delete the watchpoint you set up for this example, by right-clicking on its line in the Watchpoints window and selecting **Delete** from the pop-up menu, then close down the Watchpoints system view.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 3.4 Examining the contents of variables

Two methods of examining the contents of variables are described:

- *Contents of variables*.
  This method is simpler and shows only the contents of the specified variables.
- *Addresses and contents of variables* on page 3-9.
  This method shows the addresses of the variables as well as their contents.

### 3.4.1 Contents of variables

To examine the contents of variables as simply as possible, use the Variables processor view. In this example you start by reloading and starting the current program, then stopping it:

1. Select **Reload Current Image** from the **File** menu.
2. Select **Go** from the **Execute** menu (or press F5) to reach the first breakpoint, set by default at the beginning of function `main()`.
3. Select **Go** from the **Execute** menu (or press F5) to continue execution.
4. Enter `760` when you are prompted for the number of runs to execute. Execution continues until it reaches the breakpoint at line 150 for the 750th time. This is the breakpoint you defined in *Setting a breakpoint* on page 3-4.
5. Select **Variables** from the **Processor Views** menu if the Variables processor view is not already displayed. Reposition or resize the window if necessary. On the **Local** tab look for the `Run_Index` variable. Other variables that you can see include `Enum_Loc`, `Int_1_Loc`, `Int_2_Loc`, and `Int_3_Loc`.

   Right-click in the window, select **Properties... → Dec** and click **OK**. The display is now similar to that shown in Figure 3-5.

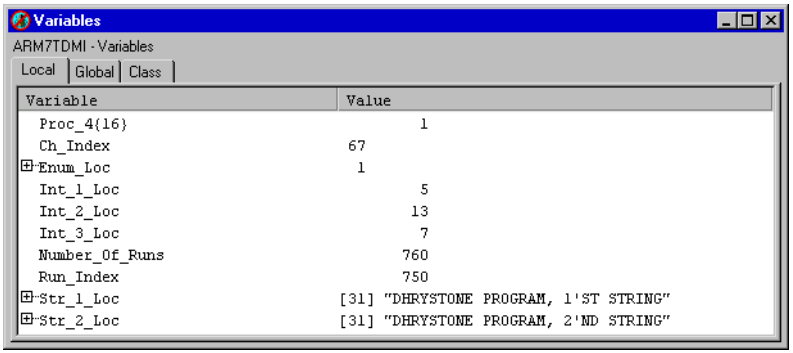



**Figure 3-5 Examining the contents of variables**


Copyright © 1999-2001 ARM Limited. All rights reserved.

6. Press F10. This is equivalent to selecting **Step** from the **Execute** menu. The program executes a single instruction and stops. Any values that have changed in the Variables processor view are displayed in red.

7. Press F10 repeatedly. As you execute the program, one instruction at a time, the values of several of the variables change. After you have allowed approximately 30 program instructions to execute, the value of `Run_Index` increases by 1. The program has now completed one further execution of the Dhrystone test.

8. Explore the various display options available from the pop-up menu. Try settings in both the **Format** submenu and the Default Display Options dialog displayed when you select **Properties...**.

   Any settings you change from **Properties...** can apply to some or all of the displayed items, depending on what is currently selected.

   For a description of the display formats available, see *Data formatting* on page 4-16.

9. Press F5 to allow the program to complete its execution, then close down the Variables processor view.

**Note**

In the Variables processor view, sub-function results are shown in the form `test_func{1}` where 1 is the result returned by calling `test_func` for the first time in this function.

### 3.4.2 Addresses and contents of variables

An alternative method of examining a variable is to use a Watch processor view. This enables you to see the memory address of the variable as well as its value. In this example you start by reloading and starting the current program, then stopping it:

1. Select **Reload Current Image** from the **File** menu.

2. Select **Go** from the **Execute** menu (or press F5) to reach the first breakpoint, set by default at the beginning of function `main()`.

3. Select **Go** from the **Execute** menu (or press F5) to continue execution.

4. Enter `760` when you are prompted for the number of runs to execute. Execution continues until it reaches the breakpoint at line 150 for the 750th time. This is the breakpoint you defined in *Setting a breakpoint* on page 3-4.

5. Select **Watch** from the **Processor Views** menu and reposition or resize the window if necessary. You can specify items to watch on several tabbed pages. In this example you examine a few variables using the first tab only.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

6. Right-click in the window, and select **Add Watch** from the pop-up menu. A Watch dialog appears, prompting you to enter an expression. For this example you enter some valid variable names, most of them preceded by an ampersand (&). See Figure 3-6.

   Enter the first expression in the Expression field by typing:

   `&Enum_Loc`

   `Enum_Loc` is a global variable, so it is stored in RAM at the address `&Enum_Loc` (these names are case-sensitive).

   **Note**

   You can also add a variable to the Watch processor view by selecting it in the source view and using the **Add Watch** pop-up menu command.

7. Press the Return key or click on the **Evaluate** button.

   The expression you entered appears in the Expression column, and its value, being the address of the variable, appears in the Value column.

   Click on the + symbol to expand the display, and another line appears showing the contents of the variable in the Value column.

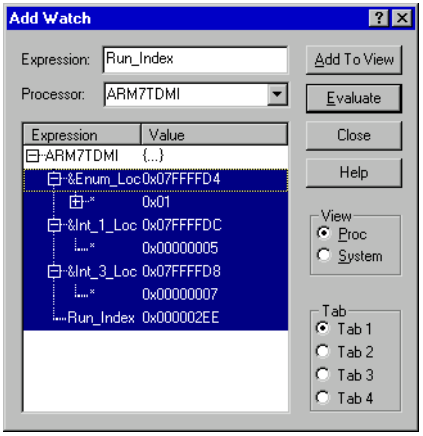



**Figure 3-6 Specifying variables to watch**

Enter, in a similar way:

`&Int_1_Loc`

`&Int_3_Loc`

`Run_Index`

Expand these lines also.


Copyright © 1999-2001 ARM Limited. All rights reserved.

The `Run_Index` variable name is not preceded by an ampersand because, in this program, the variable is stored in a processor register. Having no memory address, it is inappropriate to ask for it to be displayed. Specifying the variable name without the ampersand shows its contents but not its address.

8. Select all the lines you have entered, as shown in Figure 3-6 on page 3-10, ensure that **Proc** is the selected View and **Tab1** the selected Tab, then click the **Add To View** button and the **Close** button.

9. The variables you have specified are now displayed in the Watch processor view, and if you expand the lines you can see both the addresses and the contents of the variables.

   Move the mouse pointer to the value displayed for the `Run_Index` variable and right-click to display the pop-up menu. Select **Format** → **Decimal** so that the value of `Run_Index` is displayed as a decimal number.

10. Press F10. This is equivalent to selecting **Step** from the **Execute** menu. The program executes a single instruction and stops. Any values that have changed in the Watch processor view are displayed in color.

11. Press F10 repeatedly. As you execute the program, one instruction at a time, the values of several of the variables change. After you have allowed approximately 30 program instructions to execute, the value of `Run_Index` increases by 1. The program has now completed one further execution of the Dhrystone test.

12. Explore the various display options available from the pop-up menu. Try settings in both the **Format** submenu and the Default Display Options dialog displayed when you select **Properties...**.

    Any settings you change from **Properties...** can apply to some or all of the displayed items, depending on what is currently selected.

    For a description of the display formats available, see *Data formatting* on page 4-16.

13. Press F5 to allow the program to complete its execution, then close down the Watch processor view.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 3.5 Examining the contents of registers

To examine the contents of registers used by the currently loaded program:

1. Select **Reload Current Image** from the **File** menu.
2. Select **Go** from the **Execute** menu (or press F5) to reach the first breakpoint, set by default at the beginning of function `main()`.
3. Select **Registers** from the **Processor Views** menu and reposition or resize the window if necessary.

   The registers are arranged in groups, with only the group names visible at first. Click on the + symbol of any group name to see the registers of that group displayed, as shown in Figure 3-7.

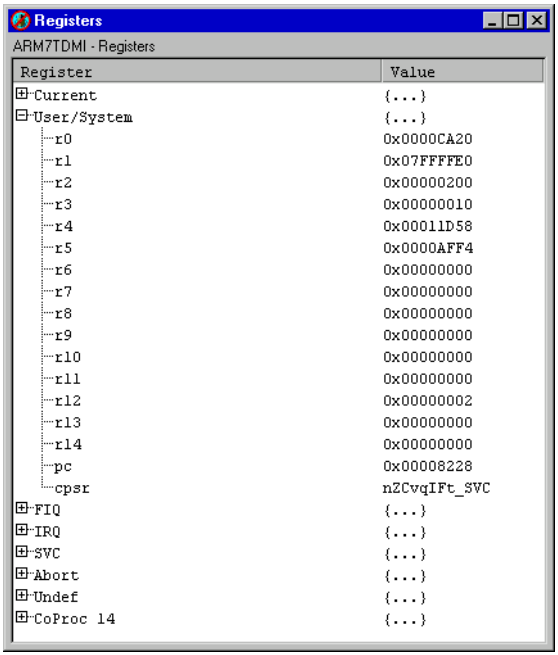



**Figure 3-7 Examining contents of registers**

4. Press F10. This is equivalent to selecting **Step** from the **Execute** menu. The program executes a single instruction and stops. Any values that have changed in the Registers processor view are displayed in red.
5. Press F10 a few more times. As you execute the program, one instruction at a time, you can see the values of several of the registers change.


Copyright © 1999-2001 ARM Limited. All rights reserved.

You soon reach the point when you are prompted, in the Console processor view, for the number of runs to perform. A very small number is sufficient this time.

6. Explore the format options available from the Registers processor view pop-up menu.

   If you position the mouse pointer on a selectable line when you right-click, the line is selected. You can change the display format of selected lines only.

   You can select multiple lines by holding down the Shift or Ctrl keys while you click on the relevant lines, in the usual way.

   For a description of the display formats available, see *Data formatting* on page 4-16.

   If you select **Add to System** from the pop-up menu, the currently selected register is added to those that are displayed in the Registers system view. This is particularly useful when your target has multiple processors and you want to examine the contents of some registers of each processor. Collecting the registers of interest into a single Registers system view avoids having to display many separate processor views.

   You can also select **Add Register** from the pop-up menu of the Registers system view. This enables you to select registers from any processor to add to those being displayed in the Registers system view.

7. Press F5 to allow the program to complete its execution, then close down the Registers processor view.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 3.6 Examining the contents of memory

To examine the contents of memory used by the currently loaded program:

1. Select **Reload Current Image** from the **File** menu.
2. Select **Go** from the **Execute** menu (or press F5) to reach the first breakpoint, set by default at the beginning of function `main()`.
3. Select **Go** from the **Execute** menu (or press F5) to continue execution.
4. Enter `760` when you are prompted for the number of runs to execute. Execution continues until it reaches the breakpoint at line 150 for the 750th time. This is the breakpoint you defined in *Setting a breakpoint* on page 3-4.
5. Select **Memory** from the **Processor Views** menu and move or resize the window if necessary. Figure 3-8 shows a typical memory processor view.

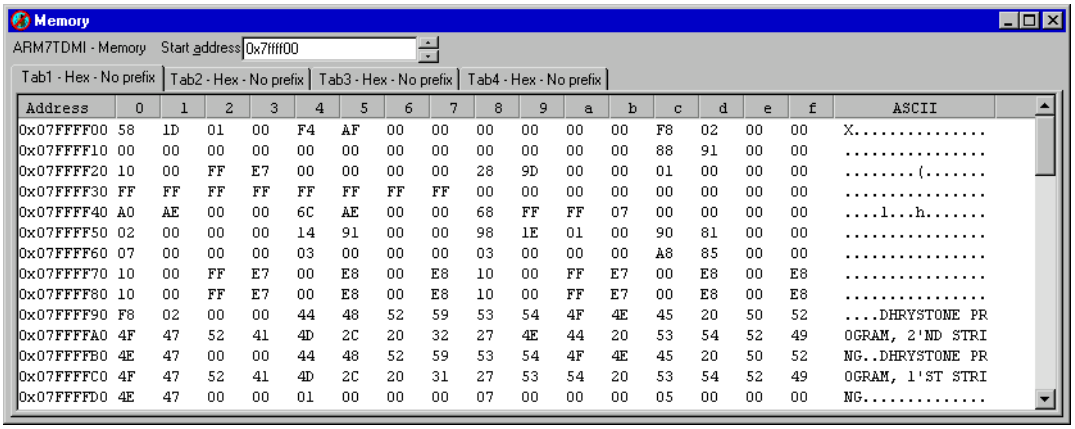



**Figure 3-8 Examining contents of memory**

*Addresses and contents of variables* on page 3-9 shows that addresses of interest are in the region of `0x07FFFFD0`, so set the Start address value to, say, `0x07FFFF00`.

6. Press F10. This is equivalent to selecting **Step** from the **Execute** menu. The program executes a single instruction and stops. Any values that have changed in the Memory processor view are displayed in red.
7. Press F10 a few more times. As you execute the program, one instruction at a time, you can see the values stored in several of the memory addresses change.


Copyright © 1999-2001 ARM Limited. All rights reserved.

8. Explore the format options available in the Memory processor view pop-up menu. Size settings appear both on the pop-up menu and in the dialog displayed when you select **Properties...** from the pop-up menu. For more information about these options see Chapter 5 *AXD Desktop*.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 3.7 Locating and changing values and verifying changes

To locate a value (of a variable or string, for example) in memory and change it:

1. Select **Reload Current Image** from the **File** menu.
2. Select **Go** from the **Execute** menu (or press F5) to reach the first breakpoint, set by default at the beginning of function `main()`.
3. Select **Memory** from the **Search** menu to display the Search Memory dialog.
4. Enter `2'ND` in the Search for field, set the In range and to addresses to `0x0` and `0xFFFF`, and select **ASCII** for the Search string type, as shown in Figure 3-9.

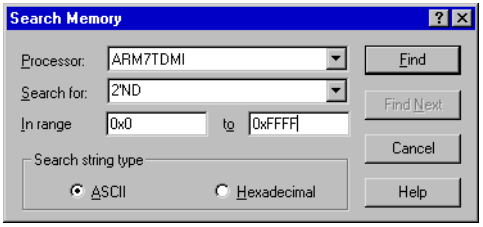



**Figure 3-9 Searching for a string in memory**

5. Click the **Find** button.
6. Click the **Cancel** button to close the Search Memory dialog.

   A Memory processor view opens if necessary, and shows the contents of an area of memory, with the string you specified highlighted. Reposition and resize the window if necessary, to see a display similar to that in Figure 3-10.

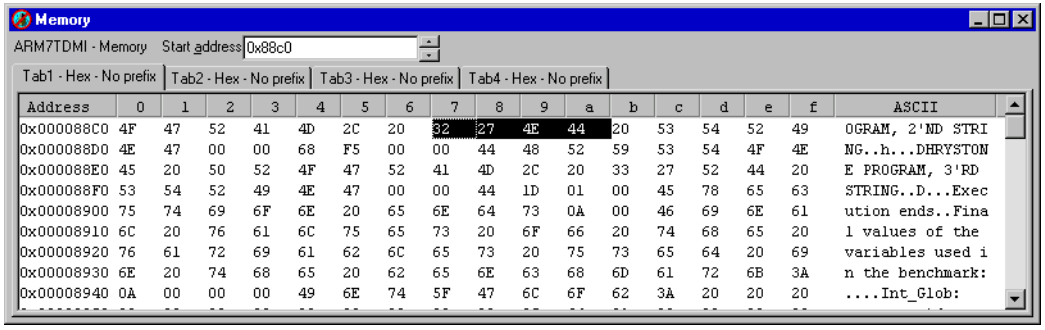



**Figure 3-10 Changing contents of memory**

You might have to right-click in the window to display the pop-up menu and set Size to `8 bit` and Format to `Hex - No prefix`.

7. The four hexadecimal values highlighted are `32 27 4E 44`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

Double-click on the value `32` and, as an example of entering a hexadecimal value, type `0x4E` and press Return.

Double-click on the value `27` and, as an example of entering an ASCII value, type `"o` (a double quote followed by a lowercase letter o) and press Return.

Double-click on the value `4E` and, as an example of entering a decimal value, type `46` and press Return.

Double-click on the value `44` and, as an example of entering an octal value, type `o62` and press Return.

8. Press F5 to continue execution, and enter a value of, say, `100` when you are prompted in the Console processor view for the number of runs to perform.

When the program displays its messages after completing its tests you can see that one of the lines that in earlier examples included the text `2'ND STRING` now has `No.2 STRING` instead because of the change you made.

In this example, the change you made was not permanent, because you did not alter the source code or the executable image stored in a disk file. You altered only the temporary copy of the image in the target memory.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 3.8 Creating a revised version of the program

In *Locating and changing values and verifying changes* on page 3-16, you tested a temporary change to your program. When developing a program you might make the same kind of temporary change and find that it is successful so must be included permanently. Showing you how to do this is beyond the scope of this book. It usually involves changes to the source code of your program, followed by recompiling and relinking. It might involve changes, not to your program, but to data received by your program.

In the simple case of the previous example, the change required to the source code is obvious. If, however, you corrected an error in execution by, say, altering the value of a variable, then the changes required in the source code might be far from obvious.

The CodeWarrior IDE enables you to make changes to source code, automate the compiling and linking processes, maintain various versions of files, and so on.

To test a new version of your program in AXD, select **Debug** from the **Project** menu of the CodeWarrior IDE.

For more information about the CodeWarrior IDE, refer to its online help or to the *CodeWarrior IDE Guide*.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

# Chapter 4
# AXD Facilities

This chapter gives a brief overview of the debugging facilities that AXD provides and contains references to sources of further information. It contains the following sections:


- *Stopping and stepping* on page 4-2
- *Expressions* on page 4-4
- *Viewing and editing* on page 4-6
- *Entering addresses* on page 4-12
- *Persistence* on page 4-13
- *RealMonitor support* on page 4-14
- *Data formatting* on page 4-16
- *Profiling* on page 4-27.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 4.1 Stopping and stepping

Ease of debugging depends on your ability to stop execution of a program at a specified point, or when specific conditions are encountered. You must then be able to examine the contents of memory, registers, or variables, possibly continue execution one instruction at a time, or specify other actions.

This section contains an overview of:

- *Breakpoints*
- *Watchpoints*
- *Stepping through a program* on page 4-3.

Detailed descriptions of how to use these facilities are given in *Execute menu* on page 5-76, and in the online help.

### 4.1.1 Breakpoints

Setting a breakpoint is the simplest way to interrupt normal execution of a program at a specific point. A breakpoint is always related to a particular memory address, regardless of what might be stored there. You set a breakpoint by specifying:

- a memory address
- a line in a listing of the executable image
- a line in the program source code that generated a program instruction
- a statement in a multi-statement line of source code
- an object, such as a low-level symbol, that indirectly specifies an address.

When execution reaches the breakpoint, normal execution stops before any instruction stored there is performed. You can then choose to examine the contents of memory, registers, or variables, or you might have specified other actions to be taken before execution resumes. In addition, any existing displays are updated to reflect the current state of the processor.

Breakpoint setting is described in *Breakpoints system view* on page 5-58, and toggling (switching on and off) in *Toggle Breakpoint* on page 5-78. You can also set breakpoints in some processor views (see *Source... processor view* on page 5-44, *Disassembly processor view* on page 5-40, and *Memory processor view* on page 5-31).

### 4.1.2 Watchpoints

A watchpoint is similar to a breakpoint, but it is the content of a watchpoint that is tested, not its address. You specify a register or a memory address to identify a location that is to have its contents tested. Watchpoints are sometimes known as data breakpoints, emphasizing that they are data dependent.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Normal execution stops if the value stored in a watchpoint changes. You might then choose to examine the contents of memory, registers, or variables, or you can specify other actions to be taken before execution resumes. In addition, any existing displays are updated to reflect the current state of the processor.

Watchpoint setting is described in *Watchpoints system view* on page 5-61.

### 4.1.3 Stepping through a program

When execution has stopped at a breakpoint or watchpoint, and you have completed your examination, you can:

- continue to the next breakpoint or watchpoint
- continue to a specific address indicated by the position of the cursor in a listing of the program image
- execute a single instruction.

If you are continuing from a call to a function, you can stop next at one of the following:

- the first executable instruction of that function
- the instruction in the calling program at which control returns from the function.

The various stepping options are described in *Execute menu* on page 5-76.

If you want to step though assembly language code you must ensure that you use frame directives in your assembly language code to describe stack usage. See the *ADS Assembler Guide* for more information.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 4.2 Expressions

This section describes:

- *Using expressions*
- *Expression rules*
- *Expression examples* on page 4-5.

### 4.2.1 Using expressions

You use expressions when you define watches in a Watch processor view or a Watch system view. An expression might be simply the name of a variable, but can, for example, involve the calculation of a memory address from the contents of various registers or variables.

Expressions are also accepted in commands you enter in the Command Line Interface view.

### 4.2.2 Expression rules

Expressions are combinations of symbols, values, unary and binary operators, and parentheses. There is a strict order of precedence in their evaluation:

1. Expressions in parentheses are evaluated first.
2. Operators are applied in precedence order.
3. Adjacent unary operators are evaluated from right to left.
4. Binary operators of equal precedence are evaluated from left to right.

AXD includes an extensive set of operators for use in expressions. Many of the operators resemble their counterparts in high-level languages such as C. There are, however, some restrictions, described in *Expression guidelines*.

#### Expression guidelines

The following rules apply to expression evaluation in AXD:

- You cannot use functions in expressions.
- You can only use C operators in constructing expressions. Any operators defined in a C++ class that also have a meaning in C (such as `[]`) do not work correctly because AXD uses the C operator instead. Specific C++ operators, such as the scope operator `::`, are not recognized.
- You cannot access base classes in standard C++ notation, for example:

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

```
class Base
{
    char *name;
    char *A;
};
class Derived : public class Base
{
    char *name;
    char *B;
    void do_sth();
};
```

If you are in method `do_sth()` you can access the member variables `A`, `name`, and `B` through the `this` pointer. For example, `this->name` returns the name defined in class `Derived`.

To access `name` in class `Base`, the standard C++ notation is:

```
void Derived::do_sth()
{
    Base::name="value"; // sets name in the base class
                        // to "value"
}
```

However, expression evaluation does not accept `this->Base::name` because AXD does not understand the scope operator. You can access this value with:

```
this->::Base.name
```

- You cannot call member functions in the form `Class::Member(...)`. This displays an error message showing that this is not a variable.
- **`private`**, **`public`**, and **`protected`** attributes are not recognized in AXD expression evaluation. This means that you can use private and protected member variables during expression evaluation because AXD treats them as public.

### 4.2.3 Expression examples

Examples of expressions that are valid in a Watch view are:

- `r3`
- `Run_Index`
- `r3 + 2 * Ch_Index`
- `Run_Index - 3 * r4`


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 4.3 Viewing and editing

When execution stops, typically at a breakpoint or watchpoint, you can view, and in some cases edit, the following types of data:

- *Control*
- *Source files*
- *Disassembled code* on page 4-7
- *Registers* on page 4-7
- *Watch* on page 4-7
- *Variables* on page 4-8
- *Memory* on page 4-8
- *Remote debug information* on page 4-10
- *High-level and low-level symbols* on page 4-10
- *Debugger internals* on page 4-10
- *Backtrace* on page 4-10
- *Debug Communications Channel* on page 4-11
- *Semihosting* on page 4-11.

The data values to be displayed are compared with the corresponding values displayed at the previous interruption of execution. Any values that have changed are displayed in color.

### 4.3.1 Control

The main Control system view provides you with information about all the objects in the current debugging session and how they interrelate. You have access to all these objects. There are four tabbed pages:

- Target
- Image
- Files
- Class.

For further information, see *Control system view* on page 5-49.

### 4.3.2 Source files

To display the source code that generated the executable code in a program image:

1. Select the **Files** tab of the Control view.
2. Expand the display of the executable image details to see the names of the source files.
3. Right-click on the file that you want to view, to display the pop-up menu.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

4. Select **Source**.
5. Right-click in the resulting view of the source file to display another pop-up menu that includes the ability to interleave disassembled code in the listing of the source file.

For further details see *Source... processor view* on page 5-44.

### 4.3.3 Disassembled code

To display disassembled code that represents a part of an executable image:

1. Select either the **Target** or the **Image** tab of the Control view.
2. Expand the display (because an image can be loaded on multiple processors), and right-click on the processor you want to examine.
3. Select **Disassembly** from the **Views** submenu of the pop-up menu.
4. Scroll to the area of code you want to examine if it is close, otherwise right-click in the Disassembly view, select **Goto...** from the pop-up menu, and specify an address in the required area.

For further details see *Disassembly processor view* on page 5-40.

### 4.3.4 Registers

To examine the registers of the current processor, select **Registers** from the **Processor Views** menu on the main menu bar.

To examine the registers in any of the target processors:

1. Select the **Target** tab of the Control view.
2. Right-click on the processor that you want to view, to display the pop-up menu.
3. Select **Registers** from the **Views** submenu.

To display a separate Registers view for each target processor, see *Registers processor view* on page 5-19. To select registers from various Registers processor views to display together in a single Registers system view, see *Registers system view* on page 5-54.

To change the value stored in any register that is displayed, double-click on its current value. In-place editing allows you to update the value.

### 4.3.5 Watch

To examine the values of specific variables or expressions related to the current processor, select **Watch** from the **Processor Views** menu on the main menu bar.

To examine specific variables or expressions related to any of the target processors:

1. Select the **Target** tab of the Control view.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

2. Right-click on the processor that you want to view, to display the pop-up menu.
3. Select **Watch** from the **Views** submenu.

You can display a separate Watch view for each available processor.

A Watch view enables you to specify expressions based on variables (from a single process) that you want to examine whenever program execution stops. This differs from a Variables view, in which only the context variables of a process are displayed.

Each Watch view has four tabbed pages for you to display expressions and their values.

Because a Watch view displays only what you have specified, the first time you open a Watch view it is empty. Right-click to display the pop-up menu. Select **Add Watch**. In the resulting Watch dialog, shown in both *Watch processor view* on page 5-23 and *Watch system view* on page 5-56, you choose which tabbed page to use and whether you are adding the new watch to a Watch processor view or a Watch system view.

You can specify expressions to be watched, but a variable name alone is often sufficient.

### 4.3.6 Variables

To examine the context variables of the current processor, select **Variables** from the **Processor Views** menu on the main menu bar.

To examine the variables in any of the available target processors:

1. Select the **Target** tab of the Control view.
2. Right-click on the processor that you want to view, to display the pop-up menu.
3. Select **Variables** from the **Views** submenu.

You can display a separate Variables view for each available processor.

Variables are defined in the executable image that you load into the memory of a target so that it can be executed by a processor. You must load an image, specifying a processor, before you can examine variables.

To change the value stored in any variable that is being displayed, double-click on its current value. In-place editing enables you to update the value.

For further details, see *Variables processor view* on page 5-26.

### 4.3.7 Memory

To examine the memory of the current processor, select **Memory** from the **Processor Views** menu on the main menu bar.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

To examine the memory in any of the available target processors:

1. Select the **Target** tab of the Control view.
2. Right-click on the processor that you want to view, to display the pop-up menu.
3. Select **Memory** from the **Views** submenu.

You can display multiple Memory views.

The four tabbed screens allow you to specify up to four areas of memory in each view. Click on a tab to bring its area of memory to the front of the display.

To change the value stored in a memory address that is being displayed, double-click on its current value. In-place editing enables you to update the value.

For further details, see *Memory processor view* on page 5-31.

### Locate using value

This provides another way for you to specify an area of memory to display. The **Locate Using Value** item is available in the pop-up menu of the following:

- registers views
- watch views
- variables views
- memory views.

In any of these views, if you select a data item that contains a memory address, then select **Locate Using Value** from the pop-up menu, a Memory view displays an area of memory that includes the specified address. For further details, see *Watch processor view* on page 5-23.

### Locate using address

This provides another way for you to specify an area of memory to display. The **Locate Using Address** item is available in the pop-up menu of the following:

- watch views
- variables views
- backtrace views
- low level symbols views.

In any of these views, if you select a data item that can be interpreted as a memory address, then select **Locate Using Address** from the pop-up menu, a Memory view displays an area of memory that includes the specified address. For further details, see *Watch processor view* on page 5-23.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 4.3.8 Remote debug information

To view low-level communication messages between the debugger and the target processor, use the **RDI Log** tab of the Output system view.

For further information, see *Output system view* on page 5-63.

### 4.3.9 High-level and low-level symbols

A high-level symbol for a procedure refers to the address of the first instruction that has been generated within the procedure, and is denoted by a function name. To see all the function names contained in an executable image, select the **Class** tab in the Control view, and expand the Globals list under the required image. Functions are marked with a colored square, and variables with a colored disc.

A low-level symbol for a procedure refers to the address that is the target for a branch instruction when execution of the procedure is required. The low-level and high-level symbols often refer to the same address.

To display a list of the low-level symbols in your program, use the Low Level Symbols processor view.

To use a low-level symbol as an expression when you define a watch, precede the symbol with `@`.

For further information, see *Entering addresses* on page 4-12 and *Low Level Symbols processor view* on page 5-35.

### 4.3.10 Debugger internals

Various internal variables contain information relevant to the current debugging session. Also, when you use ARMulator to simulate a target, statistics are accumulated during execution of the program being debugged. You can examine these statistics and information in the Debugger Internals system view which has two tabbed pages:

- Internal Variables
- Statistics (available when using a simulated target only).

For further information, see *Debugger Internals system view* on page 5-69.

### 4.3.11 Backtrace

A call stack is maintained for each processor in the target, and the Backtrace processor view enables you to examine the current state of any call stack. This shows you the path that leads from the main entry point to the currently executing function.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

All called functions are added to the stack, but those that complete execution and return control normally are removed. The stack therefore contains details of all functions that have been called but have not yet completed execution.

For further information, see *Backtrace processor view* on page 5-29.

### 4.3.12 Debug Communications Channel

The Comms Channel processor view enables you to communicate with a processor through its *Debug Communications Channel* (DCC). DCC is implemented in ARM cores containing EmbeddedICE logic. This allows low-level input and output of 32-bit words to the target. There are also facilities in the debugger to read input from a file and log output to a file.

You cannot use the Comms Channel view if DCC semihosting is being used.

For further information, see *Comms Channel processor view* on page 5-37.

### 4.3.13 Semihosting

The Console view enables you to enter data from your keyboard to the program being debugged, when it might normally receive data from some other device. You can also display on your screen output that might normally be sent elsewhere.

For further information, see *Console processor view* on page 5-39 and *Configure Processor...* on page 5-96.

If you are using Multi-ICE to connect the debugger to a target, you can select either Standard Semihosting or DCC Semihosting. You do this by setting the variable `semihosting_enabled` to a suitable value (see *Definitions* on page 6-9).

If you select DCC Semihosting, the DCC semihosting SWI handler is installed in target memory at the address specified by the `semihosting_dcchandler_address` variable. It is essential that a region of memory starting at this address is available in target memory and is unused. The default address stored in this variable is `0x70000`. You might have to change this to a lower value to suit the target memory.

**Note**

The AXD debug architecture does not currently support attaching and re-attaching while using either Standard Semihosting or DCC Semihosting.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## 4.4 Entering addresses

When you are prompted to enter an address in a field you can use:

- any of the following forms of low-level address:
  - — hexadecimal, for example:
    `0x8248` or `0x008248`
  - — decimal, for example:
    `32768`
  - — address low-level symbol such as a function name, for example:
    `@Func_2`
  - — hexadecimal address or low-level symbol, plus or minus an offset, for example:
    `@Func_2 + 0x20`
- the predefined low-level debugging symbols available in AXD, for example:
  `#pc`
- one of the following forms of high-level address:
  - — Function name, for example:
    `%Func_2`
  - — Function name and line number, for example:
    `%Func_2:164`
  - — Function name and the special symbol `$END` to signify the address one beyond the end of the function:
    `%Func_2:$END`
    The address of the last instruction in `%Func_2` can be calculated as:
    `%Func_2:$END - instructionsize`
    Where `instructionsize` is 2 for Thumb code and 4 for ARM code.
  - — A global variable, for example:
    `Int_Glob`
  - — A member of a global array, for example:
    `Arr_1_Glob[10]`


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 4.5 Persistence

You have considerable control over settings that persist from one debug session to another. By default, each debug session starts up in a state as close as possible to the final state of the previous debug session. The settings that can persist include the:

- target that was in use
- processor that was selected
- image that was loaded
- views that were open, including the size, shape, and position of each on the screen
- list of most recently used files
- list of most recently used session files
- list of most recently used images
- default display font
- tab size, specifying how tabs characters are expanded in source views
- printf formatting strings for several display formats
- array expansion threshold
- toolbar layout.

For a complete list of all the settings than can persist, see *Configure Interface...* on page 5-80.

You can also choose to restart any previous session that was saved in a session file (see *Load Session...* on page 5-12 or the `-session` argument in *AXD arguments* on page 2-3).

The current session settings are stored in the registry of your computer. The settings are also written to a file that you specify if you choose to save the current session so that you can restart it at any later time. AXD session files have the filename extension `.ses`. See *Save Session...* on page 5-13. You can create any number of session files. To restart an earlier session, see *Load Session...* on page 5-12.

If you start a debug session from the command line, specifying a session file (see *AXD arguments* on page 2-3), then that session is restarted.

If you start up AXD without specifying a session file, then the setting of the **Save and load session files** check box determines what happens. This check box is on the **General** tab of the Configure Interface dialog (see *Configure Interface...* on page 5-80). If checked, the previous session is restarted. If unchecked, system default settings are used and no earlier session is restarted.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## 4.6 RealMonitor support

RealMonitor is a software solution developed by ARM that enables you to debug a target application without stopping the processor, and without interrupting time-critical parts of the application, such as interrupt handlers.

Targets that do not incorporate RealMonitor support must stop execution when a debugger is connected, and restart when instructed to do so. The displayed views are updated each time the target execution stops. All views therefore show consistent data.

You can trace and query a RealMonitor target without interrupting its execution. This means that values of variable data displayed in AXD views might not be current. When debugging a RealMonitor target, AXD places a timestamp in the title bar of a view indicating when the contents of the view were last updated. The following restrictions apply to the timestamp value:

- The value of the timestamp is the time when the debugger makes the request to update data and does not accurately reflect (in terms of milliseconds) the time when the data are evaluated. The timestamp does indicate the order in which the views were updated.
- Views that display variable data, such as the watch, variable, and debugger internals views, often contain hierarchical data. Because these views allow unrestricted in-place expansion, they do not evaluate the child items of a structure when the structure is added to the view. This means that when a hierarchical item is expanded the child items are evaluated at the time of expansion. The values of child items might not be consistent with the timestamp.
- Evaluation of dereferenced pointers is performed when the pointer is dereferenced, not when it is added to the view. This means that the value of the dereferenced data might not be consistent with the timestamp, or with the current value of the pointer.
- Views that support dynamic addition of entries, such as the register and watch views, evaluate new entries when they are added. The existing entries and the timestamp are not updated when new entries are added. This means that the values of new entries are not consistent with the timestamp.

See *Configure Interface...* on page 5-80 for information on how to connect AXD to a RealMonitor target.

Where required, AXD views provide a **Refresh** item in their context menus that enables you to refresh the view manually. Right-click in the view to display its context menu, and select **Refresh** to update and recalculate displayed values.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

If you select **Properties...** from the Memory or Backtrace processor view, you can select or deselect an **Automatic refresh** check box. Selecting this causes the view to be refreshed automatically whenever required. This avoids you having to refresh the view manually but can impose a significant processing overhead.

**Note**

In the Variables processor view, only the currently selected tab is refreshed by the **Refresh** command. In addition, changing tabs causes the new tab to be automatically refreshed. When a tab is updated, it is marked as clean until the next step.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 4.7 Data formatting

You can enter and edit data in a variety of formats, and the debugger can display data in a variety of formats. When you enter or edit data, or view displayed data, the correct recognition of the format is vital to the interpretation of a value. It is important either to prefix a value with an indicator of the format or to be sure that the correct format is otherwise assumed.

You can select any format for displaying selected data in many places, including:

- registers views
- watch views
- variables views
- debugger internals views
- memory views.

Values are displayed in a default format for the data type unless you specify otherwise.

The **Format** item that appears on many pop-up menus leads to a number of submenus. Figure 4-1 shows a few examples.

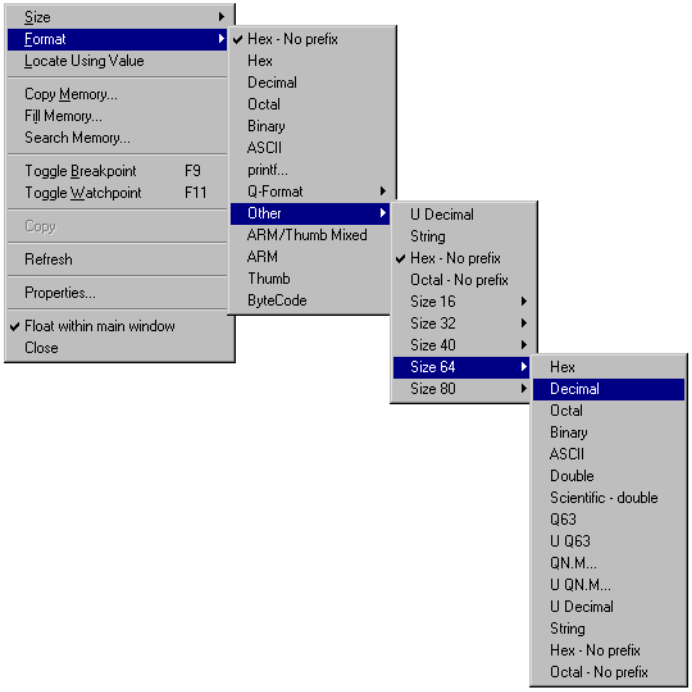



**Figure 4-1 Example format submenus**


Copyright © 1999-2001 ARM Limited. All rights reserved.

The first format listed in a format submenu is the default format for the currently selected data item. The checked format is the current format and normally appears second in the list. If the current format is the default format, then the first format is checked and is not repeated.

The format submenus allow you to select any valid data storage size and format for displaying a selected item. You might replace it with a value that you enter in yet another format.

When you enter or change data, you are offered in-place editing whenever possible, otherwise a suitable dialog is displayed. When you use in-place data editing, the whole string that you enter is interpreted and checked for validity when you press Return. If you attempt to move the focus away from the field without pressing Return, the edit is discarded and no validation occurs. Individual characters are not checked as you enter them. You can enter data in any appropriate format, not necessarily the current display format. You can enter addresses in a variety of formats, as described in *Entering addresses* on page 4-12.

If you enter a value that is larger than the available field size, the least significant bits of your value are stored and the most significant bits are ignored.

**Note**

The format of a value is often indicated by a prefix, such as `0x` meaning hexadecimal. So, to change a displayed value from `0x21` to `0x20`, for example, you must update the entry to read `0x20`. An entry of `20` is interpreted as decimal and the wrong value of `0x14` is stored.

Formats supported include:

- *Hex* on page 4-18
- *Decimal* on page 4-18
- *Octal* on page 4-18
- *Binary* on page 4-19
- *ASCII* on page 4-19
- *Printf...* on page 4-19
- *Floating point* on page 4-20
- *Registers* on page 4-21
- *Q-format* on page 4-24
- *Other* on page 4-25.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 4.7.1 Hex

A 2-character prefix of a figure `0` and an uppercase or lowercase letter `x` indicates a hex (hexadecimal) format. Hexadecimal digits (`0-9`, `A-F`) specify the value. The letters can be uppercase or lowercase. Examples are:

```
0xffff
0X000369CF
0x0
```

### 4.7.2 Decimal

A value in decimal format has no prefix. The digits `0` to `9` are allowed, and the first character can be a minus sign (`-`) or a plus sign (`+`). This format is intended for the display or entry of integer values only.

Two types of decimal format are supported:

**U Decimal** In U Decimal (unsigned decimal) format the numerical value can range from zero up to the highest value that can be stored in the number of bits available. For example, an 8-bit byte can hold values from 0 to 255.

**Decimal** In Decimal (signed decimal) format the numerical value can be negative or positive, with the maximum absolute value being half the maximum unsigned decimal value. For example, an 8-bit byte can hold values from -128 to +127 and a 16-bit halfword can hold values from -32768 to +32767.

**Note**

If you enter the value -1 this sets all the bits of the displayed, or available, format.

### 4.7.3 Octal

Octal format is generally denoted by a leading lowercase letter o. In this format each group of three bits in the stored value is represented by a digit in the range `0-7`. The grouping of bits into three starts from the least significant bit, so if the data item does not contain an exact multiple of three bits it is the most significant group that takes one or two leading zero bits for the purpose of evaluating its octal digit.

For example, you can enter or display a 16-bit halfword in octal format as, say, `o170761`. That same value is represented in binary format as `b1111000111110001` or in hexadecimal format as `0xF1F1`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 4.7.4 Binary

Binary format uses one digit, either 0 or 1, to represent each bit of a value. When you display a value in binary format, there is no leading-character indicator of the format, but the format is generally easy to recognize because it contains 0s and 1s only and is typically 8, 16, or 32 binary digits long. You can display a binary value with a space after each 4-bit nibble (see *Printf...*).

To enter a value in binary format, enter a letter `b`, in uppercase or lowercase, as the first character. You do not need to enter leading binary 0s. They are added automatically if necessary. Any spaces you enter are ignored. You can enter, for example, a space after every 4 bits to see the value of each nibble entered more clearly.

### 4.7.5 ASCII

ASCII format displays the selected data item as a fixed length string of characters. Each character represents 8 bits of storage, starting from the least significant bit. Any residual bits are padded with zeros to create a full 8 bits. The ASCII format is useful if, for example, you are examining the copying of strings and character arrays by transfer in and out of registers.

Characters displayed in ASCII format have no introductory character to indicate the format. Any non-printable value is represented by a full stop (`.`).

If you edit an ASCII character string that contains a non-printable value, the string is presented for editing in hexadecimal format.

To enter an ASCII value, prefix it with a single quotation mark (`'`) or double quotation mark (`"`). This quotation mark is not stored, it only indicates that what follows is a string of ASCII characters. Each character you enter is stored in the least significant 8 bits of the data item. Any previously entered characters shift by 8 bits to accommodate the new character. If you enter more characters than the data item can hold, the earliest characters are lost and the latest ones are stored.

### 4.7.6 Printf...

This displays a dialog allowing you to use an extended set of C formats to specify the format used for displaying the currently selected data. Examples are:

```
%d
%g
%b
%B
%f
“Hello %d world”
```


Copyright © 1999-2001 ARM Limited. All rights reserved.

If you specify a binary display format using a lowercase b, the displayed value has all its binary digits in one continuous string. Using an uppercase B in the format specification results in a display with a space after each nibble.

The last example displays a value of, say, 6 as `Hello 6 world`. If you double-click on the value to edit it, you can change just the numeric value. Changing the value from 6 to, say, 345 leads to a display of `Hello 345 world`.

For a simply-formatted value, the default printf format shown in the dialog is the format currently used to display that value.

### 4.7.7 Floating point

Most floating point formats allow you to display or enter very small and very large numerical values without having to use long strings of zeros.

You can enter very precise values by using sufficient significant digits, but the precision that can be stored depends on the number of bits allocated. Generally, if you enter too many significant digits the value is rounded to the nearest value that can be stored.

Four kinds of floating point format are supported:

**Floating point**

The first character can be a minus sign (`-`) or a plus sign (`+`). Remaining characters are decimal digits (`0-9`) and one decimal point that can be placed at any position among the digits.

A precision of up to about 6 significant figures can be stored. A value stored in this format occupies 32 bits.

**Scientific (single precision)**

A dialog helps you enter or edit data in this format. You are prompted for the sign and value of the mantissa and the exponent.

A value displayed in this format always has its decimal point after the first significant figure.

This format offers a precision of up to about 6 significant figures. The exponent value must be in the range -38 to +38. This format occupies 32 bits of storage.

**Scientific (double precision)**

A dialog helps you enter or edit data in this format. You are prompted for the sign and value of the mantissa and the exponent.

A value displayed in this format always has its decimal point after the first significant figure.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

This format offers a precision of up to about 15 significant figures. The exponent value must be in the range -308 to +308. This format occupies 64 bits of storage.

**Raw floating point**

This format enables you to view values in the 80-bit format used in a Floating Point Accelerator (FPA) coprocessor.

### 4.7.8 Registers

Certain registers contain collections of settings that can be represented by very few bits, often a single bit. Formats appropriate to these registers display the contents in meaningful ways. For example, a flag that might be on or off is displayed as a letter. The letter indicates which flag, uppercase meaning set and lowercase meaning cleared.

A dialog helps you to change the contents of this kind of register.

When you display the contents of a register, the required register format is used automatically. The following register formats are supported:

**PSR (Program Status Register)**

A typical display of a Program Status Register might show `nZCvIFtSVC`, giving information about:

- 4 condition code flags (`NZCV`)
- 2 interrupt enable flags (`IF`)
- 1 state indicator (`T`)
- 1 processor mode name (`SVC`).

**E-PSR (Enhanced Program Status Register)**

This format applies to processors, such as the ARM 9E, that support the enhanced DSP instructions in E variants of ARM Architecture version 5 and above. See *Registers processor view* on page 5-19 for more information.

A typical display of an Enhanced Program Status Register might show `nZCvqIFtSVC`, giving information about:

- 5 condition code flags (`NZCVQ`)
- 2 interrupt enable flags (`IF`)
- 1 state indicator (`T`)
- 1 processor mode name (`SVC`).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### JPSR (Jazelle Program Status Register)

This format applies to processors, such as the ARM926EJ-S, that support ARM's Jazelle™ technology for Java applications. See *Registers processor view* on page 5-19 for more information.

A typical display of a Jazelle Program Status Register might show `nZCvqIFtJSVC`, giving information about:

- 5 condition code flags (`NZCVQ`)
- 2 interrupt enable flags (`IF`)
- 2 state indicators (`TJ`)
- 1 processor mode name (`SVC`).

### FPSR (Floating Point Status Register)

This format applies if you are using a Floating Point Accelerator (FPA) coprocessor, or the Floating Point Emulator (FPE). A typical display of a Floating Point Status Register might show `xu0ZI_xuozi`.

Five letters are displayed twice. Each letter represents a floating point exception, as follows:

| | |
|---|---|
| `X` | Inexact |
| `U` | Underflow |
| `O` | Overflow |
| `Z` | Divide by zero |
| `I` | Invalid operation. |

The first set of letters represent the current settings of the five Exception Trap Enables, also known as the Exception Mask. These settings define which of the exceptions, if they occur, are intercepted by the debugger.

The second set of letters represent the Cumulative Exception Flags and show the exceptions that have occurred.

Bits 20:16 of the 32-bit FPSR are the Exception Trap Enables, and bits 4:0 are the Cumulative Exception Flags.

### FPSCR (Floating Point Status and Control Register)

This format applies to 32-bit registers containing bits that have the following meanings:

`31:28` Condition Flags. These bits represent the condition flags that contain the results of the most recent floating point comparison, as follows:

| | |
|---|---|
| `N` | the comparison produced a less than result |
| `Z` | the comparison produced an equal result |

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

- `C` the comparison produced an equal, greater than or unordered result
- `V` the comparison produced an unordered result.

`27:25` Reserved. Do not use.

`24` Mode. This bit represents the flush-to-zero mode. If the bit is unset (=0) flush-to-zero is disabled.

`23:22` Rounding mode:

- `RN` Round to Nearest
- `RP` Round towards Plus Infinity
- `RM` Round towards Minus Infinity
- `RZ` Round towards Zero.

`21:20` Stride part of the current vector length/stride control. The allowed combinations are given in the drop-down list.

`19` Reserved. Do not use.

`18:16` Length part of the current vector length/stride control. The allowed combinations are given in the drop-down list.

`15:13` Reserved. Do not use.

`12:8` Exception Mask. Each bit corresponds to one type of floating point exception, as defined for Cumulative Flags.

`7:5` Reserved. Do not use.

`4:0` Cumulative Flags. Each bit corresponds to one type of floating point exception. If a bit is set (=1), the exception flag has been set as a result of executing a floating point instruction.

The flags are defined as follows:

- `X` Inexact result, that is, non-zero rounding
- `U` Underflow has occurred
- `O` Overflow has occurred
- `Z` Divide by zero has been attempted
- `I` Invalid operation.

In views, set bits are represented by uppercase identifiers, unset bits are represented by lowercase identifiers.

### Other register formats

Other register formats might be available, depending on your target system.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 4.7.9 Q-format

Q-formats are bit-level formats for storing numeric values. A Q-format allows you to specify:

- the number of bits used to represent values
- the numeric range within which all values fall.

#### Precision

The total number of bits you allow for storing a value determines the maximum precision with which a value can be defined. You can also regard it as determining the resolution, or smallest difference that can be distinguished between values.

#### Numeric range

In a Q-format you specify how many bits represent an integer value, and how many further bits represent subdivisions within each integer value. You can in effect specify ranges. For example:

- -1024 to (almost) +1024, where the most significant 11 bits represent a signed integer value
- 0 to (almost) +64, where the most significant 6 bits represent an unsigned integer value
- -1 to (almost) +1, where the most significant bit represents the sign.

In each case, all the remaining bits represent fractions between each integer value, and (almost) means a value just one least-significant-bit less than the value given.

#### Notation

The form of a Q-format is Qn.m, where n is the number of bits before a notional *binary point*, and m is the number of bits that follow it. You can choose signed Q-format for ranges of values divided equally either side of zero, or unsigned Q-format for values that range upwards from zero.

For example, a 16-bit halfword can hold values in a signed Q4.12 format. This covers the range -8 to (almost) +8, with 65,536 unique values available in that range.

U Q15 is shorthand for unsigned Q1.15. This is a format for 16-bit values that gives 65,536 unique values in the range 0 to (almost) +2.

Q31 is shorthand for signed Q1.31. This is a format for 32-bit values that gives 4,294,967,296 unique values in the range -1 to (almost) +1.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 4.7.10 Other

This submenu includes all the remaining available formats that do not appear on other submenus. It also enables you to specify the number of bits of storage space you want allocated to a data item. The items on this submenu are:

**U Decimal** See *Decimal* on page 4-18.

**String** This format treats the value as an array of characters.

If you attempt to edit a character array formatted as a string, a String dialog is displayed, as shown in Figure 4-2.

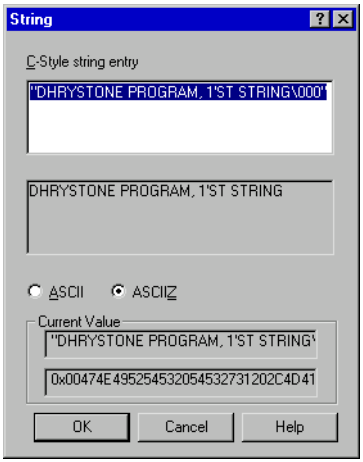



**Figure 4-2 String dialog**

You can choose whether to edit the string as ASCII or ASCIIZ. If you select ASCII, all characters up to the size of the array are replaced by the input characters. If you select ASCIIZ, a trailing character zero is always added as the final character of the array.

The dialog always opens with ASCIIZ set by default.

Type your character string in the edit box. If you enclose the string in quotes, AXD interprets it as a C++ escape string and the read-only box below the edit box shows how the string would be displayed. For example, if your input contains a null character, only the characters before the null are displayed. If you omit the quotes, all characters are treated as part of the string and it is passed directly to the display.

The current value of the string is displayed, together with its hexadecimal representation.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

**Hex - no prefix**

Select this format to display a value in hexadecimal format without the usual leading `0x` characters. If you replace the displayed value your entry must still begin with `0x` to avoid being mistaken for a decimal value.

**Octal - no prefix**

Select this format to display a value in an octal format without the usual leading `o` character. If you replace the displayed value your entry must still begin with `o` to avoid being mistaken for a decimal value.

In addition there are size menus for other sized values. These menus do not indicate the current size of the data item. This part of the submenu shows:

**Size 8** Select this to display a further submenu containing all the formats that you can use with 8-bit data items.

**Size 16** Select this to display a further submenu containing all the formats that you can use with 16-bit data items.

**Size 32** Select this to display a further submenu containing all the formats that you can use with 32-bit data items.

**Size 40** Select this to display a further submenu containing all the formats that you can use with 40-bit data items.

**Size 64** Select this to display a further submenu containing all the formats that you can use with 64-bit data items.

**Size 80** Select this to display a further submenu containing all the formats that you can use with 80-bit data items.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 4.8 Profiling

Profiling involves sampling the program counter at specific time intervals. The resulting information is used to build up a picture of the percentage of time spent in each procedure. By using the `armprof` command-line tool on the data generated by AXD, you can see how to make the program more efficient.

**Note**

Profiling is supported by ARMulator, RealMonitor, and Angel, but not by EmbeddedICE or Multi-ICE.

To collect profiling information when executing an image, you must specify profiling settings when you first load the image (see *Load Image...* on page 5-7) or before reloading the image (see *Image pop-up menu* on page 5-51):

**Flat profiling**

Flat profiling accumulates limited information without altering the image.

**Call graph profiling**

Call graph profiling accumulates more detailed information but has to add extra code to the image.

To collect profiling information:

1. Load your image file, having made the appropriate profiling settings.
2. Select **Options** → **Profiling** → **Toggle Profiling** if necessary to ensure that **Toggle Profiling** is checked in the **Profiling** submenu of the **Options** menu.
3. Execute your program.
4. Select **Options** → **Profiling** → **Write to File** when the image terminates.
5. A Save dialog appears. Enter a file name and a directory as necessary.
6. Click the **Save** button.

**Note**

You cannot display profiling information in AXD. Use the Profiling functions on the **Options** menu to capture profiling information, then use the `armprof` command-line tool, described in the *ADS Compilers and Libraries Guide*, to analyze it.

To collect information on a specific part of the execution:

1. Load (or reload) the program with profiling enabled.
2. Set a breakpoint at the beginning of the region of interest, and another at the end.
3. Execute the program as far as the beginning of the region of interest.


Copyright © 1999-2001 ARM Limited. All rights reserved.

4. Clear any profiling information already collected by selecting **Options → Profiling → Clear Collected**, and ensure that **Toggle Profiling** is checked.
5. Execute the program as far as the breakpoint at the end of the region of interest.
6. Select **Options → Profiling → Write to File** and specify the name of a file in which to save the profiling information.

You can profile both C and assembler language functions. To profile assembler language functions you must mark the functions with `FUNCTION` and `ENDFUNC` directives. See *ADS Assembler Guide* for details.


Copyright © 1999-2001 ARM Limited. All rights reserved.

# Chapter 5
# AXD Desktop

This chapter describes the menus, views, dialogs, tool and status bars that the AXD desktop provides. Chapter 2 *Getting Started in AXD* gives an overview of some of these facilities. This chapter systematically describes all the available facilities. It contains the following sections:


- *Menus, toolbars, and status bar* on page 5-2
- *File menu* on page 5-6
- *Search menu* on page 5-16
- *Processor Views menu* on page 5-18
- *System Views menu* on page 5-48
- *Execute menu* on page 5-76
- *Options menu* on page 5-80
- *Window menu* on page 5-99
- *Help menu* on page 5-102.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 5.1 Menus, toolbars, and status bar

This section introduces the AXD menus, and describes the available toolbars and the status bar.

The first screen that AXD displays is similar to that shown in Figure 5-1. Subsequent debug sessions might start up differently (see *Persistence* on page 4-13).

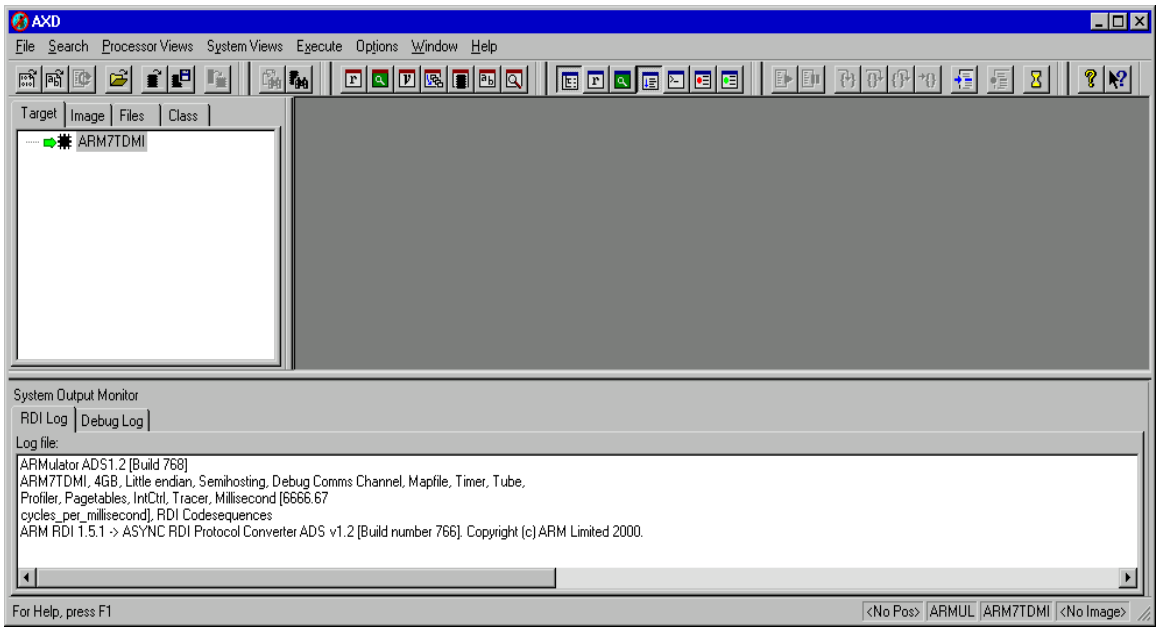



**Figure 5-1 AXD opening screen**

The main AXD features are described in this chapter under the headings:

- *Menus*
- *Toolbars* on page 5-3
- *Status bar contents* on page 5-5.

### 5.1.1 Menus

You can pull down the main menus from the menu bar near the top of the screen. Each menu in the menu bar is described in a separate section of this chapter.

Other menus, called pop-up menus, are also available when you have views displayed. Some items are duplicated in menu bar menus and pop-up menus. Some pop-up menus offer additional items. The descriptions of views in *Processor Views menu* on page 5-18 and *System Views menu* on page 5-48 include details of pop-up menus.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 5.1.2 Toolbars

Toolbars are available that correspond to most menus in the menu bar. You can display none, any, or all of these toolbars (see *Configure Interface...* on page 5-80). Clicking on an icon in a toolbar is equivalent to selecting a menu item.

#### File toolbar

File toolbar icons correspond to most **File** menu items, as shown in Figure 5-2.

**Figure 5-2 File toolbar**

These tools are described as menu items in *File menu* on page 5-6.

#### Search toolbar

Search toolbar icons correspond to most **Search** menu items, as shown in Figure 5-3.

**Figure 5-3 Search toolbar**

These tools are described as menu items in *Search menu* on page 5-16.

#### Processor Views toolbar

Processor Views toolbar icons correspond to most **Processor Views** menu items, as shown in Figure 5-4.

**Figure 5-4 Processor Views toolbar**

These tools are described as menu items in *Processor Views menu* on page 5-18.

#### System Views toolbar

System Views toolbar icons correspond to most **System Views** menu items, as shown in Figure 5-5 on page 5-4.


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Figure 5-5 System Views toolbar**

These tools are described as menu items in *System Views menu* on page 5-48.

### Execute toolbar

Execute toolbar icons correspond to most **Execute** menu items, as shown in Figure 5-6.

**Figure 5-6 Execute toolbar**

These tools, with the exception of the timed refresh tool, are described as menu items in *Execute menu* on page 5-76. For details of the timed refresh tool refer to *Window menu* on page 5-99 and *Configure Interface...* on page 5-80.

### Help toolbar

Help toolbar icons provide two ways of accessing AXD online help items, as shown in Figure 5-7.

**Figure 5-7 Help toolbar**

These tools are described in *Help menu* on page 5-102.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.1.3 Status bar contents

If you choose to display the status bar (see *Status Bar display control* on page 5-98) it appears at the bottom of the AXD screen, as shown in Figure 5-8.

**Figure 5-8 Status bar**

Help text is displayed at the left of the status bar. This either reminds you how to display information relevant to your current situation or, when you pull down a menu from the menu bar and point to an item on it, explains the purpose of that menu item.

At the right, the current debug agent, processor, and image are shown (these are not always the same as the selected debug agent, processor, and image). Also, when a source or disassembly view has the focus, the current cursor line and column are shown.

A progress indicator shows the current operation being performed by the debugger.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 5.2 File menu

**File** menu items allow you to transfer data between the debugger and various disk files, and to close down the debugger. Figure 5-9 shows the **File** menu.

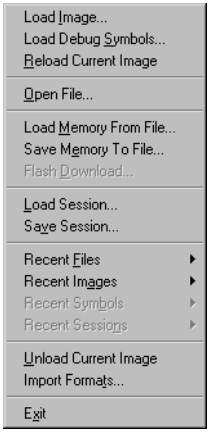



**Figure 5-9 File menu**

The **File** menu items are described under the following headings:

- *Load Image...* on page 5-7
- *Load Debug Symbols...* on page 5-8
- *Reload Current Image* on page 5-9
- *Open File...* on page 5-9
- *Load Memory From File...* on page 5-9
- *Save Memory To File...* on page 5-10
- *Flash Download...* on page 5-11
- *Load Session...* on page 5-12
- *Save Session...* on page 5-13
- *Recent Files* on page 5-13
- *Recent Images* on page 5-13
- *Recent Symbols* on page 5-14
- *Recent Sessions* on page 5-14
- *Unload Current Image* on page 5-14
- *Import Formats...* on page 5-14
- *Exit* on page 5-15.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 5.2.1 Load Image...

To select a file containing an image to load into the target memory, select **Load Image...** from the **File** menu. The resulting dialog is shown in Figure 5-10.

**Figure 5-10 Selecting an image file to load**

Navigate to the directory where the file is stored. You can specify that only files with a particular filename extension are offered for selection. The directory that you specify in this dialog becomes the current directory.

Your target might have more than one processor. The Processors list in the dialog identifies them and enables you to select those onto which you want to load the image.

Leave the **Profile** check box unchecked if you do not intend to collect any profiling information from this image. If you do want to perform profiling, then you must check the **Profile** check box and set the other profiling details in this dialog before loading the image:

**Call graph profiling**

Call graph profiling accumulates more detailed information than flat profiling but has to add extra code to the image.

**Flat profiling**

Flat profiling accumulates limited information without altering the image.


Copyright © 1999-2001 ARM Limited. All rights reserved.

When you enable profiling at load time, you are then able to start and stop the collection of profiling information during execution of the image (see *Profiling* on page 4-27).

**Note**

Profiling is supported by ARMulator, RealMonitor, and Angel, but not by EmbeddedICE or Multi-ICE.

An image loaded from the Load Image dialog or by a CLI command has a breakpoint set by default at `main()`.

If the image you are loading uses floating point data, the `$target_fpu` debugger internal variable must match the image. See *Debugger Internals system view* on page 5-69.

### 5.2.2 Load Debug Symbols...

To load only the symbols of an image onto one or more processors, select **Load Debug Symbols...** from the **File** menu. The resulting dialog is shown in Figure 5-11.

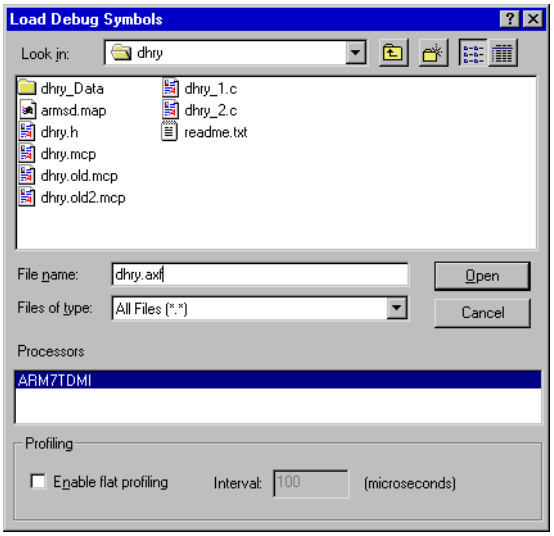



**Figure 5-11 Load Debug Symbols dialog**

Use this if the debug information is separate from the image, for example after using **Load Image From File** to load an image or if you are debugging an image in ROM.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Leave the **Enable flat profiling** check box unchecked if you do not intend to collect any profiling information from this image. When you enable profiling at load time, you are then able to start and stop the collection of profiling information during execution of the image (see *Profiling* on page 4-27).

### 5.2.3 Reload Current Image

Having finished executing an image, the simplest way of preparing it for re-execution is to reload it.

To reload the current image file, select **Reload Current Image** from the **File** menu.

You can change the profiling settings for the next execution from the Image Properties dialog (see Figure 5-65 on page 5-52).

### 5.2.4 Open File...

To examine the contents of a source file, select **Open File...** from the **File** menu. The resulting dialog is shown in Figure 5-12.

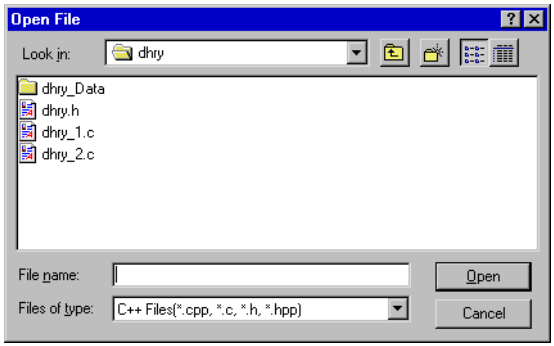



**Figure 5-12 Selecting a source file to open**

Navigate to the directory where the file is stored. You can specify that only files with a particular filename extension are offered for selection.

You can examine any source file by this means, but it does not form part of the current debugging context. Access permission is read-only, so you cannot change the contents of a source file.

### 5.2.5 Load Memory From File...

To load the contents of a file into memory, select **Load Memory From File...** from the **File** menu. The resulting dialog is similar to the one shown in Figure 5-13 on page 5-10.


Copyright © 1999-2001 ARM Limited. All rights reserved.

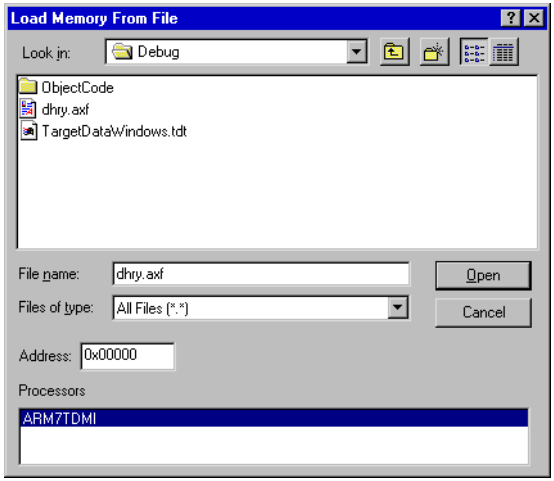



**Figure 5-13 Loading memory from file**

Specify in the Address field the memory address at which to start loading the contents of the selected file. You can enter addresses in a variety of formats, as described in *Entering addresses* on page 4-12.

### 5.2.6 Save Memory To File...

To save the contents of an area of memory to a disk file, select **Save Memory To File...** from the **File** menu. The resulting dialog is shown in Figure 5-14 on page 5-11. This dialog enables you to specify the:

- starting address of the area of memory to save
- number of bytes of memory to save
- name of a file in which to save it.

You can enter addresses in a variety of formats, as described in *Entering addresses* on page 4-12.

If more than one processor is available the Processors list identifies them and enables you to select which one is to have part of its memory saved.

Select the directory where you want to store the file containing the saved data. You can either select an existing filename or specify a new one. You also select a file type, which determines the filename extension given to any new file. If you select an existing file, the data you save overwrites the current contents of the file.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

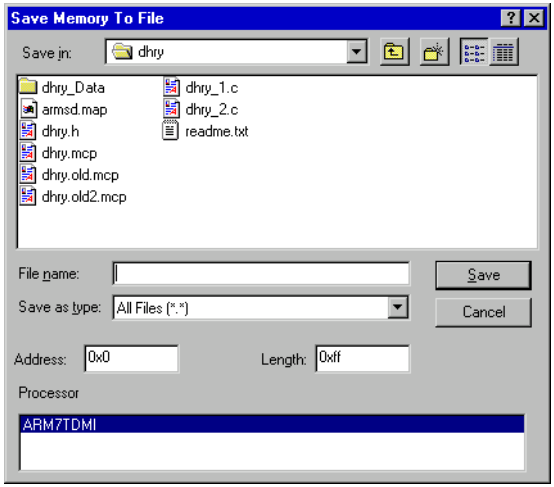



**Figure 5-14 Saving memory contents in a file**

No data conversion or formatting takes place. The file contains an exact copy of the contents of the specified memory range.

There is a limit of 16MB on the amount of memory you can specify for saving in a single file. An error message appears if the Length value you enter is too great, and you can enter a smaller value.

### 5.2.7 Flash Download...

To write an image to the Flash memory chip on an ARM Development Board or other suitably equipped hardware:

1. Select **Flash Download** from the **File** menu. The resulting dialog is shown in Figure 5-15.

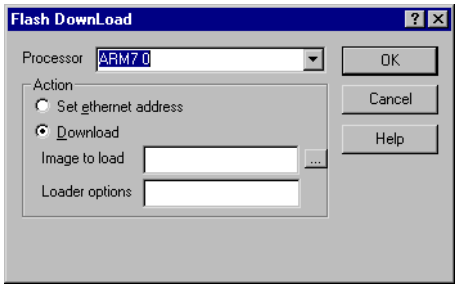



**Figure 5-15 Flash Download dialog**

2. In the Processor field, select the processor that has the Flash memory into which you want to load an image.


Copyright © 1999-2001 ARM Limited. All rights reserved.

3. In the Action group you choose either to set an Ethernet address or to download an image. Select **Download** to make a copy in Flash memory of an image stored in a file.
4. Specify in the Image To Load data entry field the file that holds the image. You can use the **Browse** button to select an image file.
5. In the Loader Options field, you can specify command-line options for the loader program.
6. When you are satisfied with all the settings, click **OK** to start the download.

If you are using Angel with Ethernet support, you can also set its Ethernet address. After writing an image to Flash memory, select **Set Ethernet Address**, click **OK**, and you are prompted for the IP address and netmask, for example 193.145.156.78. You do not have to do this if you have built your own Angel port with a fixed Ethernet address.

Refer to Appendix D *Using the Flash Downloader* for more information on Flash downloading.

### 5.2.8 Load Session...

Select **Load Session...** from the **File** menu to load a previously saved session file. The session file contains information about the state of the debugger at the time it was saved. The resulting dialog is shown in Figure 5-16.

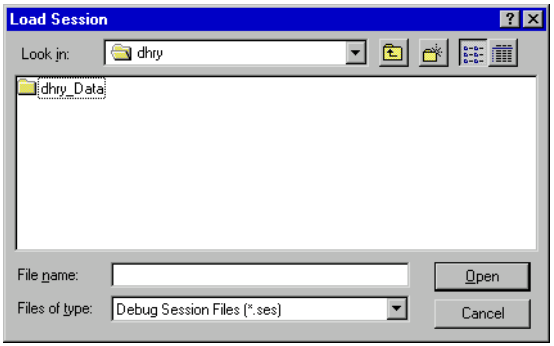



**Figure 5-16 Load Session dialog**

Locate the directory that holds the required `.ses` file, select it, and click the **Open** button.

If the session you want to resume was a recent session, the session file you require might still be in the most recently used list. See *Recent Sessions* on page 5-14.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.2.9 Save Session...

To save the current debug session so that you can reuse it at a later time, select **Save Session...** from the **File** menu. The resulting dialog is shown in Figure 5-17.

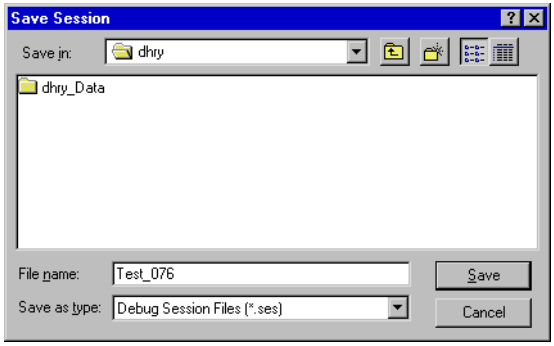



**Figure 5-17 Save Session dialog**

Change to the directory where you want to store the session file, and specify the name of the file to be written. It is usual for session files to have a `.ses` filename extension. If the file you specify already exists, you are given the choice of overwriting it or specifying another file.

### 5.2.10 Recent Files

If you have opened any files by selecting **Open File...** from the **File** menu and using the resulting browse dialog, you can reopen any of the few most recently opened more easily by selecting **Recent Files**.

A submenu lists the files you have already opened and you can click on any filename in the list to open that file again.

To change the number of files that can appear in the list, select **Options** → **Configure Interface** → **General**, set a new value for Recent File List size, and click **OK**.

### 5.2.11 Recent Images

If you have loaded any images from disk files, using the Load Image dialog, then the filenames most recently used are available to you.

To display a list of recently loaded image files, select **Recent Images**. A submenu lists the filenames and you can click on any filename in the list to load that image again.

If your target has multiple processors, a dialog is displayed allowing you to select one or more processors on which you want to load the image.


Copyright © 1999-2001 ARM Limited. All rights reserved.

To change the number of files that can appear in the list, select **Options** → **Configure Interface** → **General**, set a new value for Recent Image List size, and click **OK**.

### 5.2.12 Recent Symbols

If you have opened any symbols files by selecting **Load Debug Symbols...** from the **File** menu and using the resulting browse dialog, you can reopen any of the few most recently opened more easily by selecting **Recent Symbols**.

A submenu lists the files you have already opened and you can click on any filename in the list to open that file again.

To change the number of files that can appear in the list, select **Options** → **Configure Interface** → **General**, set a new value for Recent Symbols List size, and click **OK**.

### 5.2.13 Recent Sessions

If you have saved any earlier sessions, using the Save Session dialog, then the session files most recently used are available to you.

To display a list of recently loaded session files, select **Recent Sessions**. A submenu lists the filenames and you can click on any filename in the list to restore that session to the state it was in when it was saved.

To change the number of files that can appear in the list, select **Options** → **Configure Interface** → **General**, set a new value for Recent Session List size, and click **OK**.

### 5.2.14 Unload Current Image

To remove the current image from the target, select **Unload Current Image** from the **File** menu.

As an example, when you are debugging an image loaded in one area of memory you might want to load another image into a disjoint area of memory. The second load does not unload the first image because they can both coexist. You have to unload the first image manually.

### 5.2.15 Import Formats...

To import your own format definitions, select **Import Formats** from the **File** menu. The resulting Import Formats browse dialog enables you to locate and select a `.sdm` file. This is a supplementary display module that can include format definitions. Supplementary display modules are usually supplied by ARM Limited if required.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.2.16 Exit

To close all files and stop execution of AXD, select **Exit** from the **File** menu.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 5.3 Search menu

The **Search** menu, shown in Figure 5-18, enables you to search for specific contents, either in a source file related to a current process or in memory.

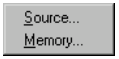



**Figure 5-18 Search menu**

The **Search** menu items are described under the following headings:

- *Source...*
- *Memory....*

### 5.3.1 Source...

To search for a given character string in a source file, select **Source...** from the **Search** menu. A dialog, shown in Figure 5-19, enables you to specify the target character string, and the file to be searched.

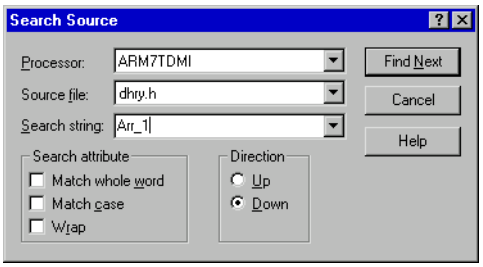



**Figure 5-19 Searching for a string in a source file**

You can search upwards or downwards, and specify case sensitivity, whether whole words only must be considered, and whether after reaching one end of the file the search continues from the other end.

When you start the search, a listing of the source file shows the lines surrounding the first occurrence of the target string, with the characters highlighted. The **Find Next** button enables you to search for the next occurrence.

### 5.3.2 Memory...

To search for a given value in memory, select **Memory...** from the **Search** menu. A dialog, shown in Figure 5-20 on page 5-17, enables you to specify what to search for and where to search.


Copyright © 1999-2001 ARM Limited. All rights reserved.

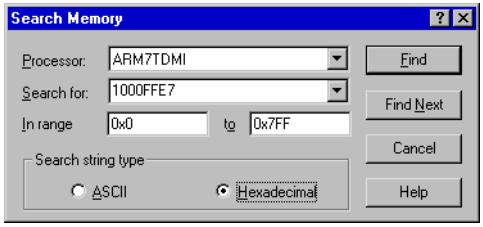



**Figure 5-20 Searching for a value in memory**

Specify the processor associated with the memory you want to search in the Processor field. The drop-down list identifies all the processors on the target and you select the one you want. Specify the first and last addresses of the area of memory you want to search in the In range and to fields. You can enter addresses in a variety of formats, as described in *Entering addresses* on page 4-12.

Specify the target value you are searching for in the Search for field. You can search for any string of up to 200 characters, using either ASCII or hexadecimal notation. Make sure you select the correct **Search string type** radio button to indicate which format you are using. The drop-down selection list contains recent search strings, making it easy for you to search again for a string you have already specified.

When you start the search, a display of the contents of memory shows the area surrounding the first occurrence of the target string, with that string highlighted. The **Find Next** button enables you to search for the next occurrence.

The value searched for is the string of bytes that you specify, in either ASCII or hexadecimal notation, and can be of any number of bytes in length. The contents of consecutive bytes of memory are compared with the target string.

**Note**

The byte order that you set (by selecting **Properties...** from the **Memory** pop-up menu) can affect the order in which bytes are displayed. This means that bytes can be displayed in a different order from that in which they are stored.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 5.4 Processor Views menu

The **Processor Views** menu, shown in Figure 5-21, enables you to examine and change information relating to specific processors.

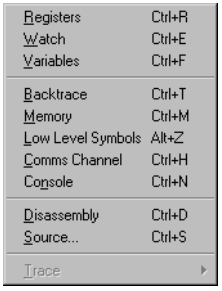



**Figure 5-21 Processor Views menu**

If you are licensed to use the *Trace Debug Tools* (TDT) and your target processor supports trace, the **Processor Views** menu also shows the **Trace** option (see *Trace processor view* on page 5-47).

All data you display and any changes you make are on the processor currently selected in the Control system view (see *Control system view* on page 5-49). The title bar of each processor view identifies the processor being viewed.

When you select a **Processor Views** menu item, a new processor view opens on the currently selected processor. If you select a processor view that is already open and displayed, it does not change. If you select a processor view that is already open and hidden, it is displayed.

You can examine one processor with any number of the available processor views. You can open a particular processor view as many times as necessary to examine all available processors. A separate viewing window appears on the screen for each view of each processor.

If you are displaying a number of processor views of the same type, with each one related to a different processor, consider using a corresponding system view instead (see *System Views menu* on page 5-48).

You can display a pop-up menu by right-clicking when the mouse pointer is inside any processor view. If the mouse pointer is on a selectable item in the view when you right-click, then that item is selected. Certain pop-up menu items are enabled only when a view item is selected, and apply to that item only.

The description of each processor view includes a reproduction of its pop-up menu. Online help gives further details.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

The **Processor Views** menu items are described under the following headings:

- *Registers processor view*
- *Watch processor view* on page 5-23
- *Variables processor view* on page 5-26
- *Backtrace processor view* on page 5-29
- *Memory processor view* on page 5-31
- *Low Level Symbols processor view* on page 5-35
- *Comms Channel processor view* on page 5-37
- *Console processor view* on page 5-39
- *Disassembly processor view* on page 5-40
- *Source... processor view* on page 5-44
- *Trace processor view* on page 5-47.

### 5.4.1 Registers processor view

The Registers processor view enables you to examine the value of any of the registers in a specific processor. It also enables you to change any of these values, unless you are debugging an Angel target when you can change the registers of the current mode only.

Ensure that the required processor is selected in the Control processor view before you display a Registers processor view. Each Registers processor view shows its processor name near the top left corner.

A typical Registers processor view is shown in Figure 5-22.

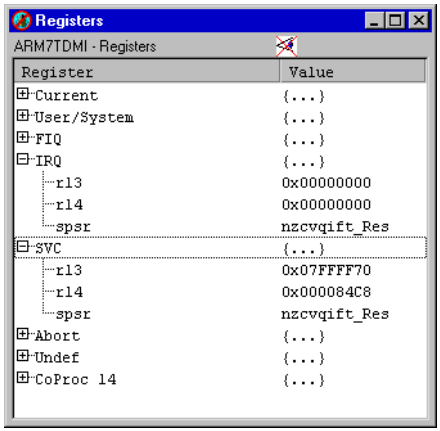



**Figure 5-22 Registers processor view**


Copyright © 1999-2001 ARM Limited. All rights reserved.

The registers are shown in named groups, to reflect the typical grouping of registers into banks. Click on the + or – boxes to expand or collapse each level of the displayed tree structure, but see *Viewing structured data* on page 2-9.

The crossed-out eye symbol is not usually present. It is displayed if you try to refresh the display of register values while the program is running, with timed refresh enabled for example, and reminds you that this is not possible. Only the Memory processor view can show values changing while the program is running.

Double-click on the value of any register that you want to change. In-place editing is invoked whenever possible, otherwise a dialog is displayed. Double-clicking on the value of a *Program Status Register* (PSR), for example, displays the dialog shown in Figure 5-23.

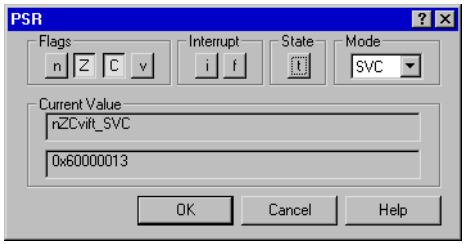



**Figure 5-23 Program Status Register dialog**

ARM processors that have an extra bit (Q, signifying saturation) in the program status register require an *Enhanced PSR* (EPSR) format. This displays the extra bit in the Registers processor view, and editing the value of that register displays the dialog shown in Figure 5-24.

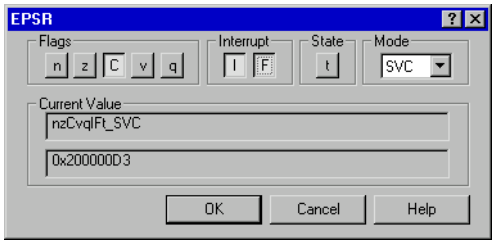



**Figure 5-24 Enhanced Program Status Register dialog**

ARM processors that are Jazelle-capable have an extra bit (J, signifying Jazelle state) in the program status register and require a *Jazelle PSR* (JPSR) format. This displays the extra bit in the Registers processor view, and editing the value of that register displays the dialog shown in Figure 5-25 on page 5-21.


Copyright © 1999-2001 ARM Limited. All rights reserved.

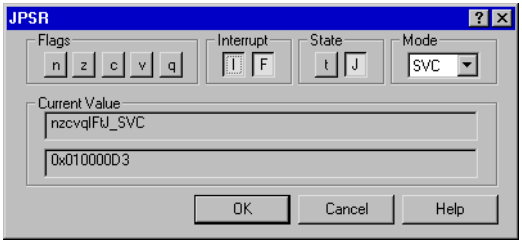



**Figure 5-25 Jazelle Program Status Register dialog**

Whenever AXD can determine the most suitable format for displaying a program status register, it does so automatically. If AXD is unable to determine the most suitable format, EPSR is used by default. To change the display format for a program status register, select one from the **Registers** submenu of the **Format** menu item in the Registers processor view pop-up menu.

For more information about data display formats and data entry formats, see *Data formatting* on page 4-16.

To add one of the registers displayed in a Registers processor view to the Registers system view (see *Registers system view* on page 5-54), right-click on the required register to select it and display the pop-up menu, then select **Add to System** (see *Registers processor view pop-up menu*).

### Registers processor view pop-up menu

To display the Registers pop-up menu, shown in Figure 5-26, right-click within the Registers processor view.

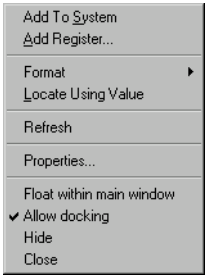



**Figure 5-26 Registers processor view pop-up menu**


Copyright © 1999-2001 ARM Limited. All rights reserved.

The **Add To System**, **Format**, and **Locate Using Value** menu items are enabled only when you right-click on a selectable item in the processor view, and then they apply to the selected item only.

**Format** Select **Format** to see a list of all the available formats in which you can display the item currently selected in the Registers processor view, as shown in Figure 5-27.

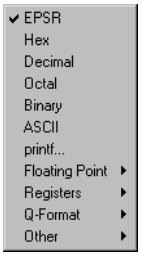



**Figure 5-27 Formats available for displaying registers**

Refer to *Data formatting* on page 4-16 for details of the formats available.

**Locate Using Value**

The **Locate Using Value** menu item functions as described in *Watch processor view pop-up menu* on page 5-24.

**Refresh** Select **Refresh** to update and recalculate the displayed data values. A Registers processor view cannot be refreshed while an image is executing. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

**Properties...** Select **Properties...** to display the Default Display Options dialog shown in Figure 5-28 on page 5-23.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

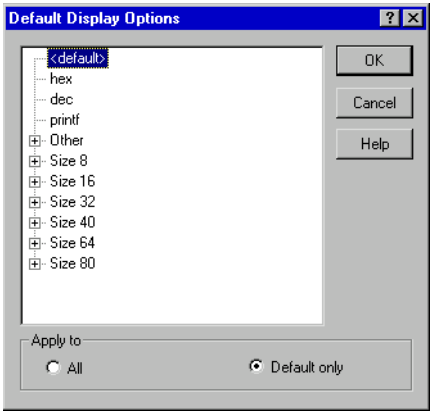



**Figure 5-28 Default Display Options dialog**

With this dialog you control the default display format, and choose whether any change you make applies to all the displayed data items or to only those that currently use the default format. Click the **Help** button in the dialog to display more information.

If you hide a Registers processor view then later select it again, it reappears in the state it was in when you hid it.

If you close a Registers processor view then later select it again, it is displayed as though you are selecting it for the first time.

### 5.4.2 Watch processor view

The Watch processor view enables you to examine the value of variables, or of expressions dependent on variables, in an image being executed by a specific processor.

Select the required processor in the Control system view before you display a Watch processor view. Each Watch processor view shows its processor name near the top left corner.

A Watch processor view is initially empty. You choose what is to be listed and have its value shown. One way to add lines to a Watch processor view is to select one or more items in a Variables processor view, then right-click and select **Add to Processor Watch** from the resulting pop-up menu.

Another way to add a line is to select **Add Watch** from the pop-up menu (see Figure 5-30 on page 5-24). Your specification of what is to be watched is shown in the first column, and its value is evaluated and shown in the second column each time program execution in the relevant processor stops. (When using certain processors, execution does not have to stop. See *RealMonitor support* on page 4-14.)

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

To define what is to be watched, you enter an expression. An expression can be simply the name of a variable, and that is often all you require. More complex expressions are allowed, however, and might include logical and arithmetic operators, in addition to the names of variables and constants.

If the displayed data has a tree structure, click on the + or – boxes to expand or collapse each level of the structure, but see *Viewing structured data* on page 2-9.

A typical Watch processor view is shown in Figure 5-29. For more information about data display formats and data entry formats, see *Data formatting* on page 4-16.

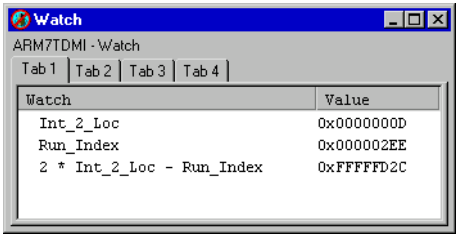



**Figure 5-29 Watch processor view**

The four tabbed pages allow you to define up to four lists of expressions to watch in any one processor. Click on the tab of the page you want to view.

### Watch processor view pop-up menu

To display the Watch pop-up menu, shown in Figure 5-30, right-click within the Watch processor view.

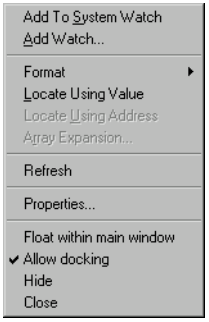



**Figure 5-30 Watch processor view pop-up menu**

If you have selected an item in the Watch processor view, you can click on **Add to System Watch** to add that item to those displayed in a Watch system view (see *Watch system view* on page 5-56).


Copyright © 1999-2001 ARM Limited. All rights reserved.

One way of defining a new watch for the Watch processor view is to select **Add Watch** from the pop-up menu. The resulting dialog is shown in Figure 5-31.

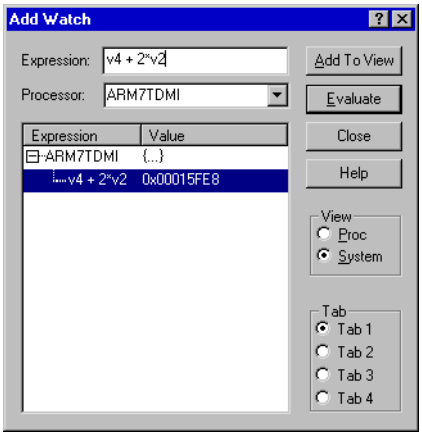



**Figure 5-31 Add Watch dialog**

Enter a new expression to watch. Specify the processor, whether the new watch must be added to the Watch processor view or system view (see *Watch system view* on page 5-56), and on which tabbed page it must appear. Figure 5-31 shows **Tab 1** of the Watch processor view as the chosen destination. By default, the **Tab** radio button selected reflects the current tabbed page in the Watch processor view.

Click the **Evaluate** button to evaluate the expression. Either the result of the evaluation or an error message appears in the main pane of the dialog. You can build up a list of expressions and their values. Select any one of the displayed expressions and click the **Add To View** button to add that expression to the specified view (Watch processor view or Watch system view). To see the address of a variable in addition to its value, enter & in front of its name.

From the Watch processor view (as shown in Figure 5-29 on page 5-24) you can select a data item and use the Watch processor view pop-up menu to examine it in more detail.

**Locate Using Value**

Select **Locate Using Value** or **Locate Using Address** if you want to examine an area of memory. Selecting this option means that only the 32 least significant bits of the value of the selected item are used as the required memory address. A Memory Locate view is displayed, very similar in appearance to the view described in *Memory processor view* on page 5-31, with the selected memory address in view. If an existing tabbed page in a Memory processor view already includes the required


Copyright © 1999-2001 ARM Limited. All rights reserved.

address (and is not the page from which the request originates), that page is displayed. If no existing tabbed page is suitable, the least recently selected tabbed page is used to display the required region of memory.

**Locate Using Address**

Selecting this option is similar to **Locate Using Value** and enables you to examine an area of memory in a Memory Locate view. Selecting this option, however, uses the address of the selected item as the required memory address.

**Array Expansion...**

Select **Array Expansion...** to display an Array Expansion dialog, either when you are about to expand an array or when you want to display a different range of elements in an array that is already expanded. This dialog enables you to choose either to display all elements or to specify the first and last element numbers to display. Array elements are numbered from zero. A 50-element array, for example, contains elements numbered 0 to 49. By default, elements 0 to 15 (the first 16 elements) only are displayed when you expand any array with more than 16 elements.

**Refresh** Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

**Properties...**

Select **Properties...** to display the Default Display Options dialog shown in Figure 5-28 on page 5-23. With this dialog you control the default display format, and choose whether any change you make applies to all the displayed data items or to only those that currently use the default format. Click the **Help** button in the dialog to display more information.

If you hide a Watch processor view then later select it again, it reappears in the state it was in when you hid it.

If you close a Watch processor view then later select it again, it is displayed empty, as though you are selecting it for the first time.

### 5.4.3 Variables processor view

The Variables processor view enables you to examine and change the value of any of the listed variables.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Click on the appropriate tab to display:

- **Local** variables, those with scope within the current function
- **Global** variables, those with scope over all parts of the program
- **Class** variables, those with scope within the current class only.

A Variables processor view is shown in Figure 5-32, with its **Local** tab selected.

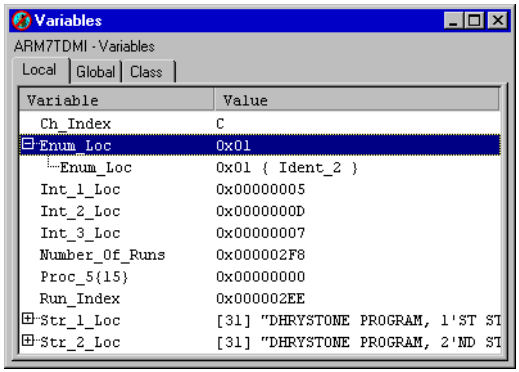



**Figure 5-32 Variables processor view**

Click on the + or – boxes to expand or collapse each level of the displayed tree structure, see *Viewing structured data* on page 2-9.

Double-click on the value of any variable that you want to change. In-place editing is invoked whenever possible, otherwise a dialog is displayed. For more information about data display formats and data entry formats, see *Data formatting* on page 4-16.

### Variables processor view pop-up menu

To display the Variables pop-up menu, shown in Figure 5-33 on page 5-28, right-click within the Variables processor view.


Copyright © 1999-2001 ARM Limited. All rights reserved.

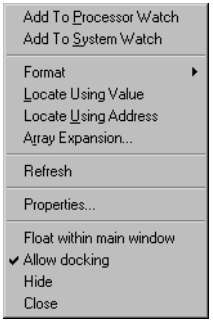



**Figure 5-33 Variables processor view pop-up menu**

If the mouse pointer is on a selectable line when you right-click, then that line is selected. The items in the top group of the pop-up menu apply to the selected line only. If no line is selected, those items are disabled.

You can select more than one displayed item by left-clicking while holding down the Shift or Ctrl key. A right-click on one of the selected items then displays the pop-up menu and any resulting actions apply to all the selected items.

The **Locate Using Value**, **Locate Using Address**, and **Array Expansion...** menu items function as described in *Watch processor view pop-up menu* on page 5-24.

**Add To Processor Watch**

Select **Add To Processor Watch** to add the selected variable(s) to a Watch processor view (see *Watch processor view* on page 5-23).

**Add To System Watch**

Select **Add To System Watch** to add the selected variable(s) to a Watch system view (see *Watch system view* on page 5-56).

**Refresh** Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

**Properties...** Select **Properties...** to display the Default Display Options dialog shown in Figure 5-28 on page 5-23. With this dialog you control the default display format, and choose whether any change you make applies to all the displayed data items or to only those that currently use the default format. Click the **Help** button in the dialog to display more information.

If you hide a Variables processor view then later select it again, it reappears if possible with the same tab selected and the same levels expanded as when you hid it. The content depends on the current execution context (the address stored in the program counter).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

If you close a Variables processor view then later select it again, it is displayed as though you are selecting it for the first time.

### 5.4.4 Backtrace processor view

The Backtrace processor view enables you to examine the call stack of the current image in a specific processor.

Select the required processor in the Control system view before you display a Backtrace processor view. Each Backtrace processor view shows its processor name near the top left corner.

A typical Backtrace processor view is shown in Figure 5-34.

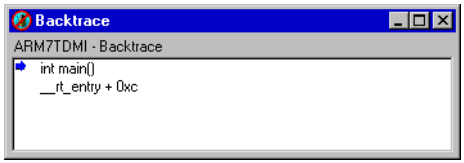



**Figure 5-34 Backtrace processor view**

Each entry in the displayed list shows the function context of a single stack frame. The entries are ordered with the current stack frame at the top. An entry contains the address or the name of a function, and the types of the parameters with which it was called. An address is displayed instead of a name if the address is not in a range described by a symbol table or image.

It is possible for an application program to overwrite and damage the call stack. A line showing ----------//----------//---------- indicates that an inconsistency has been detected and the call stack is considered broken. This might be due, for example, to the use of inline calls.

A stack discontinuity can also result from a call to another image if the debug symbol table of the called image is not available to the debugger. A call to an operating system function is an example. You can display a complete call stack if you first load the debug symbol tables of all the images your program calls. See *Load Debug Symbols...* on page 5-8.

When the selected processor is in Jazelle state, the Backtrace processor view contains only a single entry, shown in Figure 5-35 on page 5-30. This entry is given as an offset from the nearest previous low-level symbol.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

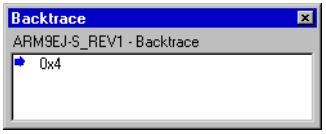



**Figure 5-35 Backtrace processor view in Jazelle state**

### Backtrace processor view pop-up menu

To display the Backtrace pop-up menu, shown in Figure 5-36, right-click within the Backtrace processor view.

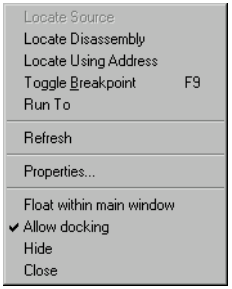



**Figure 5-36 Backtrace processor view pop-up menu**

If the mouse pointer is on a selectable line when you right-click, then the line is selected. The items in the top group of the pop-up menu apply to the selected line only. If no line is selected, those items are disabled.

The **Locate Using Address** menu item functions as described in *Watch processor view pop-up menu* on page 5-24.

Select **Refresh** to refresh the call stack. This is necessary only when **Automatic Refresh** is unselected in the Backtrace Properties dialog. If **Automatic Refresh** is selected, the call stack is refreshed automatically but this can impose a significant processing overhead.

To display the dialog shown in Figure 5-37 on page 5-31, select **Properties...** from the pop-up menu.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

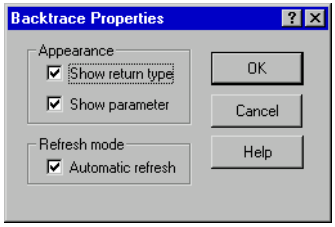



**Figure 5-37 Backtrace Properties dialog**

Refer to AXD online help for details of the other pop-up menu items.

### 5.4.5 Memory processor view

The Memory processor view enables you to examine and change the contents of specific memory addresses.

Memory is made available to you in pages. The default size of a page is 1024 bytes, but you can change this value by selecting **Properties...** from the Memory pop-up menu.

The area of memory visible depends on the size that you make the processor view window. If less than one page of memory is visible, scroll bars allow you to view other parts of the current page. A typical view of an area of memory is shown in Figure 5-38.

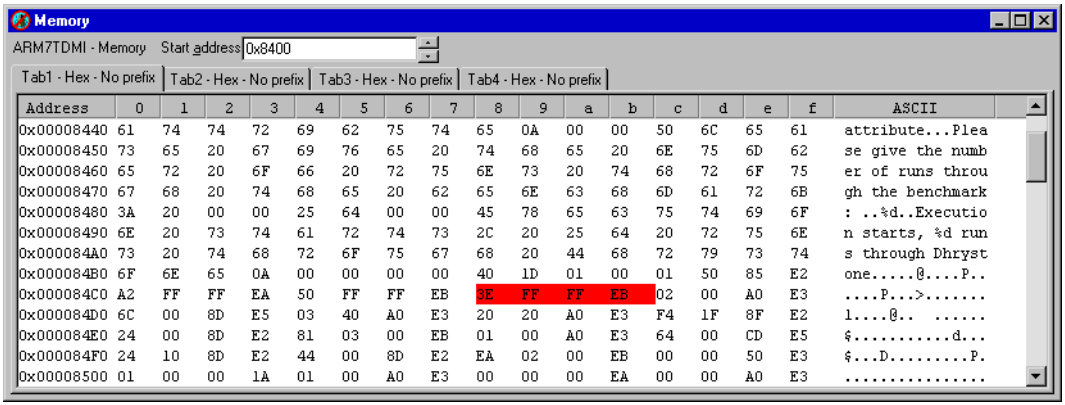



**Figure 5-38 Memory processor view**

You can specify the Start address in a variety of formats, as described in *Entering addresses* on page 4-12.

Generally, each line represents 16 bytes of memory. The address of the first byte is shown at the left. Using **Properties...** from the Memory pop-up menu, you can set this to be either the absolute address or the zero-based offset from the beginning of the

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

current page. The contents of the 16 bytes of memory occupy most of each line. You can display these as four 32-bit words, eight 16-bit half-words, or sixteen 8-bit bytes. In the last case, the ASCII characters corresponding to the 16 bytes are shown at the right of the line.

The four tabbed pages allow you to define up to four memory areas of interest and to switch easily from one to another. The memory area covered by each tabbed page is one page long, and starts at the address you specify in the Start Address field near the top of the view. The areas you define can overlap, or be contiguous, or be separate.

The size of the displayed words and their display format are among the settings you can change using the Memory processor view pop-up menu. You can use different settings on each of the four tabbed pages of the view. The column widths change automatically to suit the format you select. If you specify a printf format without specifying a width parameter, then the display uses a column width of 10 characters plus any decoration characters you specify.

A breakpoint is highlighted in red, or in gray-red if it is disabled. A watchpoint is highlighted in green, or gray-green if it is disabled.

The Memory processor view also enables you to examine memory on a processor running in Jazelle state, shown in Figure 5-39 on page 5-33. You can set the display format for the first tabbed page to `ByteCode` using the Memory pop-up menu.

You can open multiple memory views, even on a single processor, if you want more than four tabbed pages. For more information about data display formats and data entry formats, see *Data formatting* on page 4-16.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

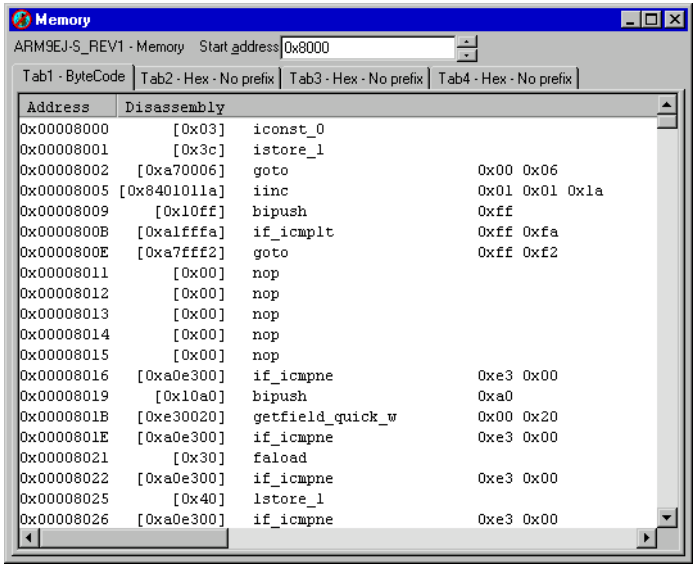



**Figure 5-39 Memory processor view in Jazelle state**

## Memory processor view pop-up menu

To display the Memory pop-up menu, shown in Figure 5-40, right-click within the Memory processor view.

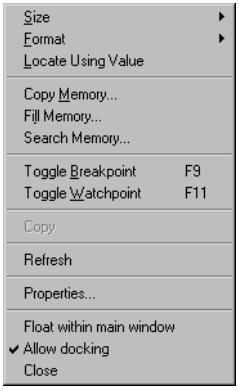



**Figure 5-40 Memory processor view pop-up menu**


Copyright © 1999-2001 ARM Limited. All rights reserved.

The **Locate Using Value** menu item functions as described in *Watch processor view pop-up menu* on page 5-24.

**Format** Select **Format** from the Memory processor view pop-up menu to display the submenu, shown in Figure 5-41, to set the display format. You can use different settings on each of the four tabbed pages of the view.

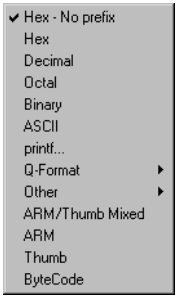



**Figure 5-41 Memory processor view formats**

**Toggle Breakpoint**

Select **Toggle Breakpoint** to toggle a breakpoint at the address defined by the current cursor position. If a breakpoint already exists at this address it is deleted. If no breakpoint exists at this address a default breakpoint is created here.

**Toggle Watchpoint**

Select **Toggle Watchpoint** to toggle a watchpoint at the address defined by the current cursor position. If a watchpoint already exists at this address it is deleted. If no watchpoint exists at this address a default watchpoint is created here.

A new watchpoint set in this way from the Memory processor view can watch for changes in the value stored in one or more bytes of memory. If the tabbed page of the Memory processor view is configured to display 8-bit, 16-bit, or 32-bit values, then 1, 2, or 4 bytes respectively are watched. If a block of memory locations is selected when you create a new watchpoint with **Toggle Watchpoint**, then all the highlighted locations are watched.

**Refresh** Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

Refer to AXD online help for details of the other Memory pop-up menu items, including the Memory Properties dialog, shown in Figure 5-42 on page 5-35.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

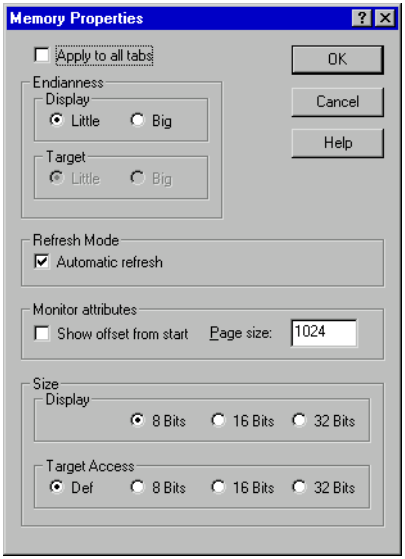



**Figure 5-42 Memory Properties dialog**

#### Data width for memory reads and writes

The Target Access group of radio buttons in the Memory Properties dialog enables you to specify the width of data read from or written to memory. Unless you have a particular requirement, use the **Def** setting to indicate that you want the debugger to decide.

### 5.4.6 Low Level Symbols processor view

The Low Level Symbols processor view enables you to examine the low-level symbols of the current image in a specific processor.

Select the required processor in the Control system view before you display a Low Level Symbols processor view. Each Low Level Symbols processor view shows its processor name near the top left corner.

A typical Low Level Symbols processor view is shown in Figure 5-43 on page 5-36.


Copyright © 1999-2001 ARM Limited. All rights reserved.

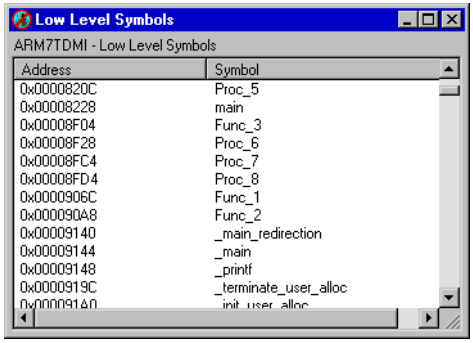



**Figure 5-43 Low Level Symbols processor view**

The left column shows addresses and the right column shows symbol strings. Use the pop-up menu, or click on the column heading, to sort the list by address order or by symbol name order.

If you hide a Low Level Symbols processor view then later select it again, it reappears in the state it was in when you hid it.

If you close a Low Level Symbols processor view then later select it again, it is displayed as though you are selecting it for the first time.

### Low Level symbols processor view pop-up menu

To display the Low Level Symbols pop-up menu, shown in Figure 5-44, right-click within the Low Level Symbols processor view.

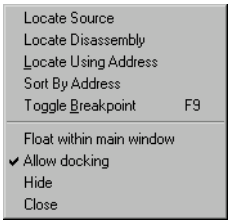



**Figure 5-44 Low Level Symbols processor view pop-up menu**

If the mouse pointer is on a selectable line when you right-click, then that line is selected. The items in the top group of the pop-up menu apply to the selected line only. If no line is selected, those items are disabled.

The **Locate Using Address** menu item functions as described in *Watch processor view pop-up menu* on page 5-24.

 *Copyright © 1999-2001 ARM Limited. All rights reserved.* 

Refer to AXD online help for more details of these menu items.

### 5.4.7 Comms Channel processor view

The Comms Channel processor view enables you to examine data that passes to and from the debugger target along the *Debug Communications Channel* (DCC), and to send data of your own. AXD has its own built-in DCC viewer. If your target offers, for example, the file `ThumbCV.dll` as a DCC viewer, do not select it.

The Comms Channel Viewer processor view is shown in Figure 5-45. You can enable or disable the debug communications channel by checking or clearing a check box on the Comms Channel Properties dialog available from the Comms Channel Viewer pop-up menu (see *Comms Channel Viewer pop-up menu* on page 5-38) or on the Processor Properties dialog (see *Configure Processor...* on page 5-96 or *Control system view pop-up menus* on page 5-50).

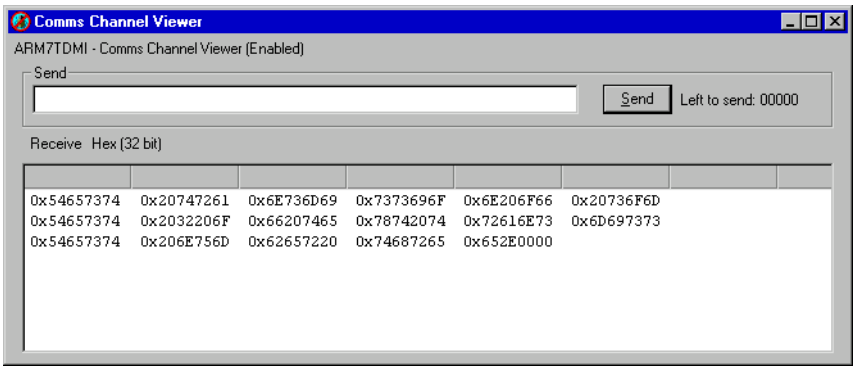



**Figure 5-45 Comms Channel Viewer processor view**

Use the Send group of this window to send information down the channel. Type information in the edit box and click the **Send** button to store the information in a buffer. The information is sent when requested by the target, in ASCII character codes. The Left to send counter displays the number of bytes that are left in the buffer.

By default, the information received by the Comms Channel Viewer is displayed using the Auto-Toggle format. This converts the information into ASCII character codes and displays it in the Receive pane, if the channel viewer is active.

However, if `0xFFFFFFFF` is received, the Auto-Toggle format displays the following words as a hexadecimal number. On the next occurrence of `0xFFFFFFFF` the Auto-Toggle format switches and information is again converted into ASCII character codes. In this way, `0xFFFFFFFF` is used to toggle between the different formats.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

You can display received information in other formats, as described in *Comms Channel Viewer pop-up menu*.

### Comms Channel Viewer pop-up menu

To display the Comms Channel Viewer pop-up menu (see Figure 5-46), right-click anywhere in the Comms Channel processor view except in the Send edit area or the Receive pane.

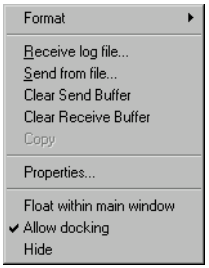



**Figure 5-46 Comms Channel Viewer pop-up menu**

**Receive log file...**

Select this option to specify a file where data received from the target is stored.

**Send from file...**

Select this option to specify a text file where data is stored ready to be sent to the target.

**Clear Send Buffer**

Select this option to flush the send buffer and to close an input file if used.

**Format** Select this option to display the submenu shown in Figure 5-47.

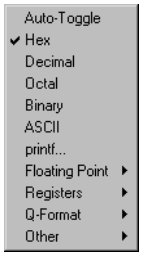



**Figure 5-47 Comms Channel Viewer formats**

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

When you select any format except **Auto-Toggle** (the default), information received is shown in columns. Your choice of format determines the initial column width, but you can change the column widths by using the mouse to drag the column header dividers to the left or right.

**Properties...** Select **Properties...** from the Comms Channel Viewer pop-up menu to display the dialog shown in Figure 5-48.

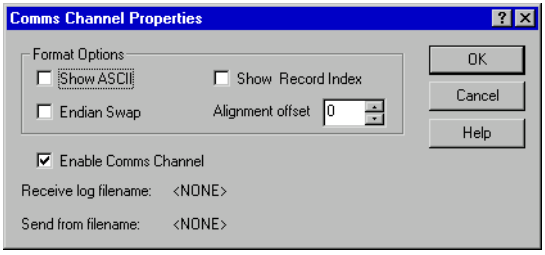



**Figure 5-48 Comms Channel Properties dialog**

The Comms Channel Properties dialog is used to enable the Comms Channel Viewer and to specify how information from the Comms Channel is displayed.

AXD online help describes this and all the Comms Channel Viewer pop-up menu items.

### 5.4.8 Console processor view

You might want to debug an image that is intended to receive input from, or write output to, devices that are not yet available. The Console processor view provides the semihosting facility that enables you to do so.

Output from an executing image is displayed, and you can respond by entering data from your keyboard or from a file to provide input for the image.

A typical Console processor view is shown in Figure 5-49.

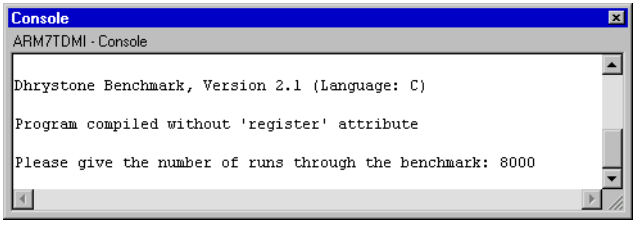



**Figure 5-49 Console processor view**


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Console processor view pop-up menu

To display the Console processor view pop-up menu, shown in Figure 5-50, right-click within the Console processor view.

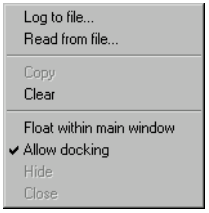



**Figure 5-50 Console processor view pop-up menu**

Refer to AXD online help for more details of these menu items.

## 5.4.9 Disassembly processor view

The Disassembly processor view displays not only the contents of regions of memory but also the assembler code instructions that correspond to those contents.

A typical Disassembly processor view is shown in Figure 5-51. This is the display format you see if you have both **Show margin** and **Show addresses** selected on the Properties dialog obtained from the pop-up menu (see Figure 5-55 on page 5-43).

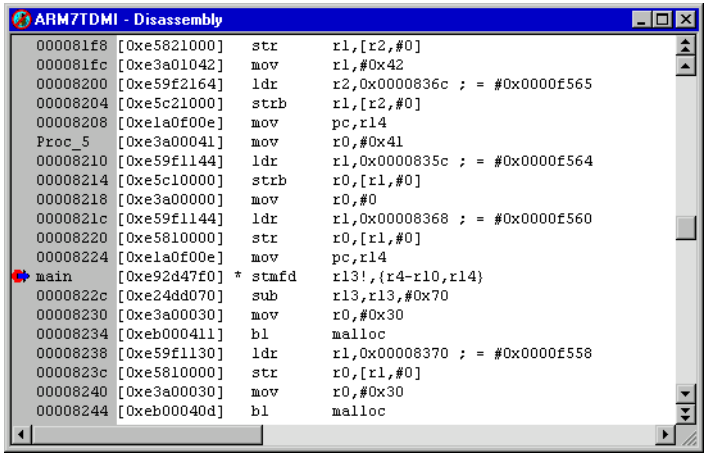



**Figure 5-51 Disassembly processor view**

You can see the low-level symbols in the margin and a blue arrow shows the current execution point. Any breakpoints are marked with a red disc.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

A Disassembly processor view for a Jazelle-capable processor is shown in Figure 5-52. This is the display format you see if you select **ByteCode** from the **Disassembly Mode** option on the pop-up menu (see Figure 5-54 on page 5-43).

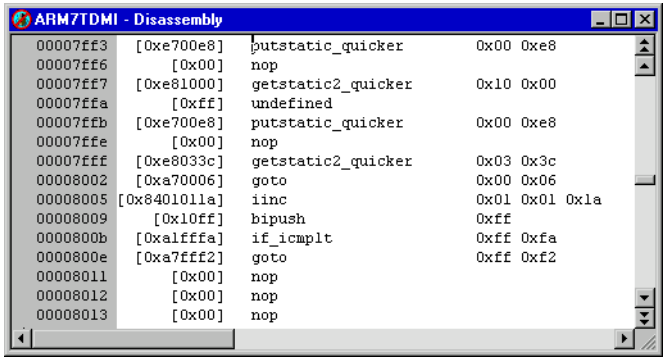



**Figure 5-52 Disassembly processor view in Jazelle state**

When running your image on a Jazelle-capable processor, there are circumstances that might result in erroneous disassembly. This is because, unlike ARM or Thumb code, bytecodes are of variable length and so disassembly can become unsynchronized. In this case low-level symbols might not be displayed correctly where the disassembly is out of synch.

**Note**

If an area of erroneous disassembly includes the execution address then the blue arrow indicator is not shown in the Disassembly processor view.

### Disassembly processor view pop-up menu

To display the Disassembly pop-up menu, shown in Figure 5-53 on page 5-42, right-click within the Disassembly processor view.


Copyright © 1999-2001 ARM Limited. All rights reserved.

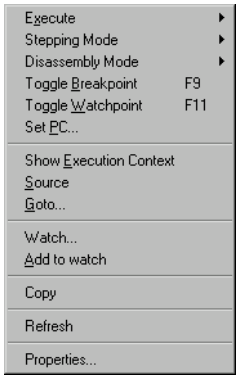



**Figure 5-53 Disassembly processor view pop-up menu**

To display a submenu duplicating the items you are most likely to want from the **Execute** main menu, select **Execute** on the pop-up menu. See *Execute menu* on page 5-76 for details of all but one of these items.

**Set Next Statement** is the item that appears on the **Execute** submenu and not in the **Execute** main menu. To resume execution at a specific statement, without executing any intervening statements, right-click on the required statement in the Disassembly processor view, select **Execute** in the pop-up menu, and select **Set Next Statement**.

To display a submenu allowing you to change the setting of the stepping mode, select **Stepping Mode** on the pop-up menu. The stepping modes available are:

**Disassembly** This steps always in disassembly instructions.

**Strong Source** This steps always in source code statements.

**Weak Source** This steps in source code statements if possible. This is the default setting.

If the image contains no debug information, stepping is by disassembly instructions.

If the image contains debug information, but the source files are not accessible, stepping is by instructions corresponding to source lines.

To display a submenu allowing you to change the setting of the code used for disassembly, select **Disassembly Mode** on the pop-up menu, shown in Figure 5-54 on page 5-43. This enables you to display the disassembled code in ARM/Thumb, or ARM, or Thumb, or ByteCode format. If you choose ARM/Thumb then AXD displays the code depending on what the image contains.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

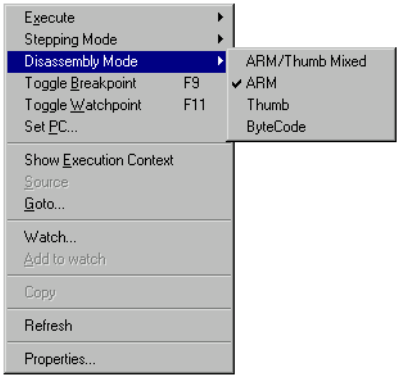



**Figure 5-54 Disassembly Mode formats submenu**

**Toggle Breakpoint**

Select this option to set or delete a breakpoint at the current cursor position.

**Toggle Watchpoint**

Select this option to set or delete a watchpoint at the current cursor position.

**Set PC** Select this option to reset the program counter so that the instruction at the current cursor position is the next instruction to be executed.

**Refresh** Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

**Properties...** Select this option to display the View Properties dialog shown in Figure 5-55.

Refer to AXD online help for more details of all the items on the pop-up menu.

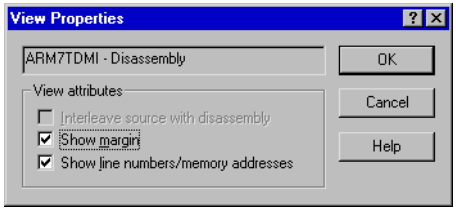



**Figure 5-55 Disassembly View Properties dialog**

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 5.4.10 Source... processor view

The Source... processor view first displays a file selection dialog, similar to that shown in Figure 5-56.

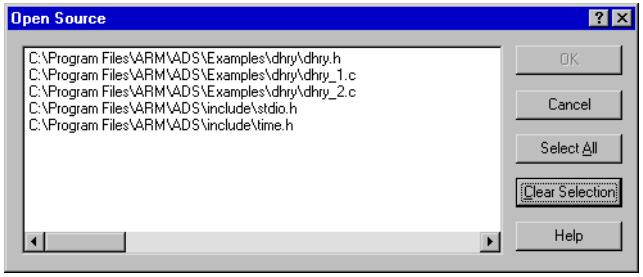



**Figure 5-56 Source file selection**

This lists all source files that have contributed debug information to the current image (not necessarily all source files used to build the image). Select a filename and click the **OK** button to display the file. If the file is not in the expected place, another dialog enables you to specify where it is or browse to find it.

The Source... processor view displays the source file as specified. To display the Source... processor view pop-up menu, shown in Figure 5-58 on page 5-45, right-click within the view.

Figure 5-57 on page 5-45 shows the kind of source file listing you see if you select **Interleave disassembly** from the pop-up menu.

You can set a breakpoint by double-clicking on a line number or address at the left side of the display, or by right-clicking in a line and selecting **Toggle Breakpoint** from the pop-up menu. You can set a breakpoint on a procedure exit by double-clicking on the line number of the line containing the closing bracket of the procedure.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

```
ARM7TDMI - C:\Program Files\ARM\ADS\Examples\dhry\dhry_1.c
000084a4 [0x67756f72]   dcd      0x67756f72  roug
000084a8 [0x68442068]   dcd      0x68442068  h Dh
000084ac [0x74737972]   dcd      0x74737972  ryst
000084b0 [0x0a656e6f]   dcd      0x0a656e6f  one.
000084b4 [0x00000000]   dcd      0x00000000  ....
000084b8 [0x00011d40]   dcd      0x00011d40  @...
000084bc [0xe2855001]   add      r5,r5,#1
000084c0 [0xeaffffa2]   b        0x8350  ; (main + 0x128)
147        {
148
149          Proc_5();
000084c4 [0xebffff50]   bl       Proc_5
150          Proc_4();
000084c8 [0xebffff3e] * bl       Proc_4
151            /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
152          Int_1_Loc = 2;
000084cc [0xe3a00002]   mov      r0,#2
000084d0 [0xe58d006c]   str      r0,[r13,#0x6c]
153          Int_2_Loc = 3;
000084d4 [0xe3a04003]   mov      r4,#3
154          strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
000084d8 [0xe3a02020]   mov      r2,#0x20
000084dc [0xe28f1ff4]   add      r1,pc,#0x3d0 ; #0x88b4
000084e0 [0xe28d0024]   add      r0,r13,#0x24
000084e4 [0xeb000381]   bl       __rt_memcpy_w
155          Enum_Loc = Ident_2;
000084e8 [0xe3a00001]   mov      r0,#1
000084ec [0xe5cd0064]   strb     r0,[r13,#0x64]
```

**Figure 5-57 Source... processor view**

Interleaved disassembly code sometimes includes lines containing just six dots (.....). These lines indicate breaks in the sequence of execution due to inline code expansion and compiler optimization settings. The memory address displayed at the beginning of each line helps you to see how your source code is compiled.

### Source... processor view pop-up menu

To display the pop-up menu shown in Figure 5-58, right-click within the Source... processor view.

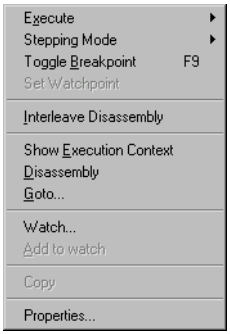



**Figure 5-58 Source... processor view pop-up menu**


Copyright © 1999-2001 ARM Limited. All rights reserved.

To display a submenu duplicating the items you are most likely to require from the **Execute** main menu, select **Execute** on the pop-up menu. See *Execute menu* on page 5-76 for details of all items except **Set Next Statement**.

**Set Next Statement** is the item that appears on the **Execute** submenu and not in the **Execute** main menu. To resume execution at a specific statement, without executing any intervening statements, right-click on the required statement in the Source... processor view, select **Execute** in the pop-up menu, and select **Set Next Statement**.

To display a submenu allowing you to change the setting of the stepping mode, select **Stepping Mode** on the pop-up menu. The stepping modes available are:

**Disassembly** This steps always in disassembly instructions.

**Strong Source** This steps always in source code statements.

**Weak Source** This steps in source code statements if possible. This is the default setting.

If the image contains no debug information, stepping is by disassembly instructions.

If the image contains debug information, but the source files are not accessible, stepping is by instructions corresponding to source lines.

To activate or deactivate a breakpoint at the current cursor position, select **Toggle Breakpoint** from the pop-up menu. To set or replace a watchpoint on a currently selected item, select **Set Watchpoint** from the pop-up menu.

To display the dialog shown in Figure 5-59, select **Properties...** from the pop-up menu.

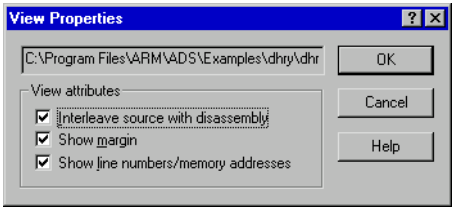



**Figure 5-59 Source View Properties dialog**

Refer to AXD online help for more details of all the items on this pop-up menu.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 5.4.11 Trace processor view

The *Trace Debug Tools* (TDT) is part of the ARM RealTime Trace solution for debugging and can be purchased as an extension to ADS. If you are licensed to use this product, and your target processor supports trace, the **Trace** option is also available on the **Processor Views** menu.

For further information see the documentation accompanying the product, for example, the *Trace Debug Tools User Guide*.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 5.5 System Views menu

System views are not specific to any processor. Some show information about the whole system. Others help you reduce the number of views you need to display.

A Registers system view, for example, can show registers that are associated with several processors. You can examine in a single system view registers that otherwise require multiple processor views. In a system view, the processor to which each line is related is identified in the display.

Selecting a **System Views** menu item generally toggles that view. That is, the selected system view is opened if it is currently closed or hidden, or hidden if it is currently open. System views that are open are checked on the menu. Figure 5-60 shows an example of a **System Views** menu.

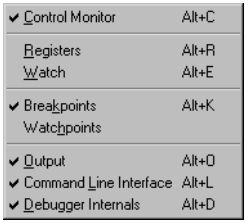



**Figure 5-60 System Views menu**

Each system view has a pop-up menu you can display by right-clicking when the mouse pointer is inside the system view. If the mouse pointer is on a selectable line in the system view when you right-click, then that line is selected. Certain pop-up menu items are enabled only when a line is selected, and apply to that line only.

The description of each system view includes a reproduction of its pop-up menu. Online help gives further details.

The **System Views** menu items are described under the following headings:

- *Control system view* on page 5-49
- *Registers system view* on page 5-54
- *Watch system view* on page 5-56
- *Breakpoints system view* on page 5-58
- *Watchpoints system view* on page 5-61
- *Output system view* on page 5-63
- *Command Line Interface system view* on page 5-65
- *Debugger Internals system view* on page 5-69.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 5.5.1 Control system view

The Control system view shows details of all current processors, and enables you to examine this information in several ways. Tabbed pages available are:

- Target
- Image
- Files
- Class.

Figure 5-61 shows a Control system view with its **Files** tab selected.

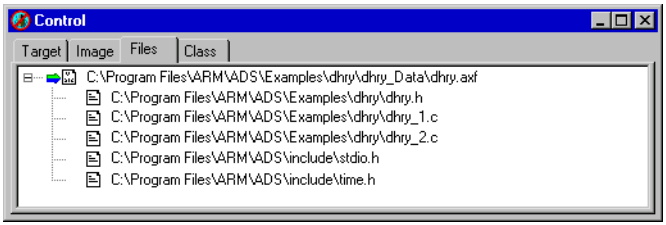



**Figure 5-61 Control system view**

Expand or collapse each level of the displayed tree structure by clicking on the + or – boxes.

The tabbed pages contain the following information:

**Target** Lists the processors on the target. Where a processor has an associated module, such as a coprocessor or ETM, the processor entry is expanded and the module is shown as a child. You cannot expand a child entry any further.

One processor can be designated the current processor. If so, it is indicated by a green arrow in the display. Commands you issue apply by default to the current processor. For example, when you select an item from a menu in the main menu bar it applies to the current processor.

One processor can be designated the selected processor. If so it is indicated by being highlighted in blue in the display. You select a processor by clicking on its name. When you select a menu item from a pop-up menu it applies to the selected processor.

Whenever possible, the current processor is the selected processor.

**Image** Lists the images loaded in the memory of the target. Expand an image node to show the processor with which the image is associated.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

One image can be designated the current image. If so, it is indicated by a green arrow in the display. Commands you issue apply by default to the current image. For example, when you select an item from a menu in the main menu bar it applies to the current image.

One processor can be designated the selected image. If so it is indicated by being highlighted in blue in the display. You select an image by clicking on its name. When you select a menu item from a pop-up menu it applies to the selected image.

**Files** Lists the files associated with all the images on the target. Expand an image node to show the files associated with that image.

**Class** Lists the classes associated with all the images on the target. Expand an image node to show a globals node, and a class node if the image contains any class information. Expand the globals node to show a list of global functions and global variables. Expand a class node to show a list of classes contained in the image. Expand a class to show a list of member functions and member variables.

### Control system view pop-up menus

When you right-click in a Control system view, the pop-up menu that appears depends on which tabbed page is currently selected and which item on that page is currently selected.

The items you can select on each tabbed page are as follows:

- on the **Target** tab, you can select a processor
- on the **Image** tab, you can select an image or a processor
- on the **Files** tab, you can select an image or a file
- on the **Class** tab, you can select an image, a function, or a variable.

If the mouse pointer is on a selectable line when you right-click, then that line is selected. Any pop-up menu items that do not apply to the selected line are disabled. Some of the pop-up menu items are equivalent to menu items from the menu bar.

Brief details follow of the Control pop-up menus. AXD online help gives more details.

#### *Processor pop-up menu*

With a processor selected, the pop-up menu is as shown in Figure 5-62 on page 5-51.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

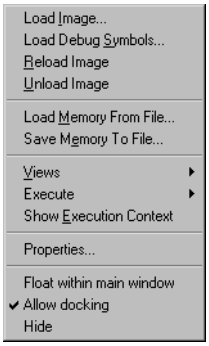



**Figure 5-62 Pop-up menu when a processor is selected**

Select **Properties...** from this pop-up menu to display the dialog shown in Figure 5-63.

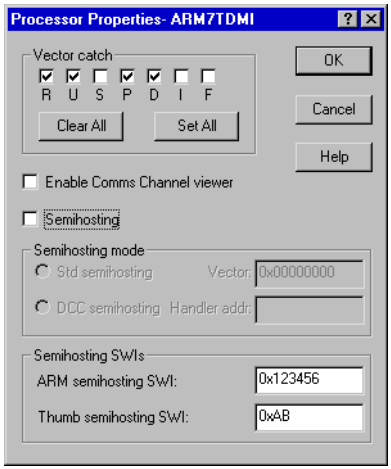



**Figure 5-63 Processor Properties dialog**

For a description of this dialog, see *Configure Processor...* on page 5-96.

### *Image pop-up menu*

With an image selected, the pop-up menu is as shown in Figure 5-64 on page 5-52.


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Figure 5-64 Pop-up menu when an image is selected**

If you select **Properties...** from this pop-up menu, the dialog shown in Figure 5-65 is displayed.

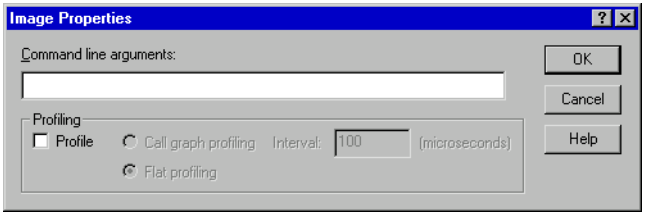



**Figure 5-65 Image Properties dialog**

The Image Properties dialog enables you to specify Command-line arguments. These are the arguments you supply if you start execution of the image by entering a command at a command-line prompt. They are supplied to the program when you load, or reload, and execute it in AXD.

The Image Properties dialog also shows the Profiling settings that become effective the next time you load or reload an image. You can change these settings to be those you want when the next image execution begins. The settings shown are not necessarily those currently in force, because you might have changed them since the last load or reload operation.

### *File pop-up menu*

With a file selected, the pop-up menu is as shown in Figure 5-66.

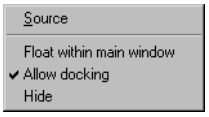



**Figure 5-66 Pop-up menu when a file is selected**

Select **Source** from this pop-up menu to display a Source processor view, showing the source code associated with the selected file.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Function pop-up menu

With a function selected, the pop-up menu is as shown in Figure 5-67.

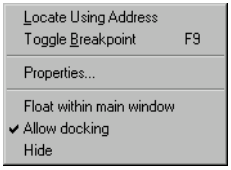



**Figure 5-67 Pop-up menu when a function is selected**

Select **Properties...** from this pop-up menu to display the dialog shown in Figure 5-68.

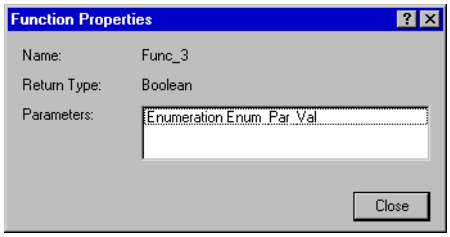



**Figure 5-68 Function Properties dialog**

The Function Properties dialog shows the name and type of the function, and the parameters that it takes.

### Variable pop-up menu

With a variable selected, the pop-up menu is as shown in Figure 5-69.

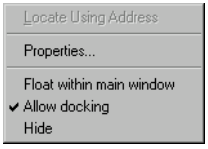



**Figure 5-69 Pop-up menu when a variable is selected**

If you select **Properties...** from this pop-up menu, the dialog shown in Figure 5-70 on page 5-54 is displayed.


Copyright © 1999-2001 ARM Limited. All rights reserved.

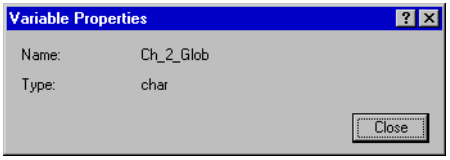



**Figure 5-70 Variable Properties dialog**

The Variable Properties dialog shows the name and type of the variable.

### 5.5.2 Registers system view

The Registers system view can display registers from more than one processor. It also enables you to change any of these values, unless you are debugging an Angel target when you can change the registers of the current mode only.

If you want to see the values of a few registers in various processors change as your program executes, you can display the registers in a single Registers system view. This can avoid displaying a number of Registers processor views.

The registers are displayed in groups, under processor names and register bank names. Click on the + or – boxes to expand or collapse each level of the displayed tree structure, but see *Viewing structured data* on page 2-9.

Figure 5-71 shows a typical Registers system view.

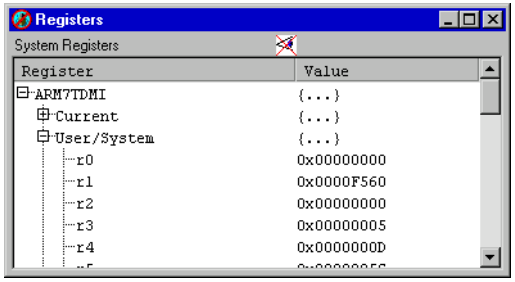



**Figure 5-71 Registers system view**

Double-click on the value of any register that you want to change. In-place editing is invoked whenever possible, otherwise a dialog is displayed.

The crossed-out eye symbol is not usually present. It is displayed if you try to refresh the display of register values while the program is running, with timed refresh enabled for example, and reminds you that this is not possible. Only the Memory processor view can show values changing while the program is running.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Registers system view pop-up menu

To display the Registers pop-up menu, shown in Figure 5-72, right-click within the Registers system view.

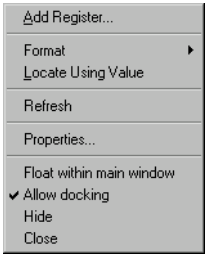



**Figure 5-72 Registers system view pop-up menu**

**Add Register**

To add a register from any processor to those displayed in a Registers system view, select **Add Register** from the pop-up menu.

**Format**

If you right-click on a register line, it is selected. The **Format** menu item is enabled when a register line is selected, and applies to the selected line only.

Refer to *Data formatting* on page 4-16 for details of the formats available, and to AXD online help for other details of the Registers pop-up menu items.

**Locate Using Value**

The **Locate Using Value** menu item functions as described in *Watch processor view pop-up menu* on page 5-24.

**Refresh**

Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

If you hide a Registers system view then select it, it reappears in the state it was in when you hid it.

If you close a Registers system view then select it, it is displayed empty, as though you are selecting it for the first time.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.5.3 Watch system view

The Watch system view enables you to examine the value of variables, or of expressions depending on variables, in the images associated with various processors. You might require several processor views to see what you can display in a single system view.

A Watch system view is initially empty. You specify expressions. These expressions are evaluated each time program execution stops, and the values displayed. One way to add lines to this view is to select one or more items in a Variables processor view, then right-click and select **Add to System View** from the resulting pop-up menu.

Another way to add a line to the Watch system view is to select **Add Watch** from its pop-up menu to display an Add Watch dialog (see Figure 5-75 on page 5-57).

An expression can be simply the name of a variable. Expressions can also include logical and arithmetic operators in addition to the names of variables and constants. If the displayed data has a tree structure, click on the + or – boxes to expand or collapse each level of the structure, but see *Viewing structured data* on page 2-9.

A typical Watch system view is shown in Figure 5-73.

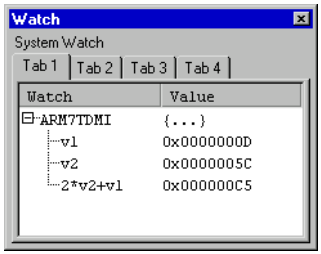



**Figure 5-73 Watch system view**

You can define lists of expressions to watch on up to four tabbed pages. Click the tab of a page to display it.

If you hide a Watch system view then select it, the view reappears in the state it was in when you hid it.

If you close a Watch system view then select it, the view is displayed empty, as though you are selecting it for the first time.

#### Watch system view pop-up menu

When you right-click in a Watch system view, the pop-up menu that appears depends on which item on that page is currently selected.


Copyright © 1999-2001 ARM Limited. All rights reserved.

If the mouse pointer is on a selectable line when you right-click, then that line is selected. Any pop-up menu items that do not apply to the selected line are disabled.

To display the Watch pop-up menu, shown in Figure 5-74, right-click within the Watch system view.

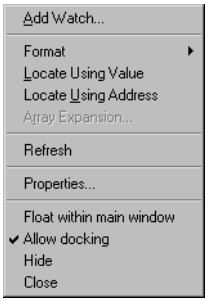



**Figure 5-74 Watch system view pop-up menu**

To display the dialog shown in Figure 5-75, select **Add Watch** from the pop-up menu.

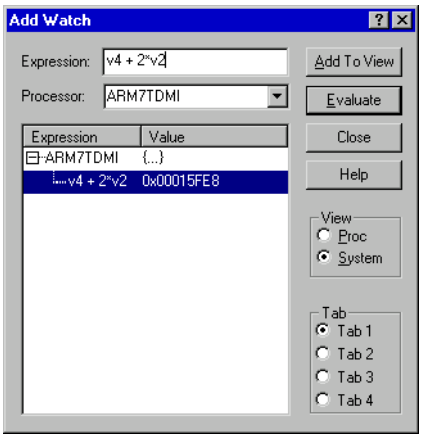



**Figure 5-75 Add Watch dialog**

Enter a new expression to watch. Specify the processor, whether the new watch must be added to the Watch processor view or system view, and on which tabbed page it must appear. Figure 5-75 shows **Tab 1** of the Watch system view as the chosen destination. By default, the **Tab** radio button selected reflects the current tabbed page in the Watch system view. Select an expression and click the **Evaluate** button to see the result of its evaluation.

To add the selected expression to the chosen view, click the **Add To View** button.


Copyright © 1999-2001 ARM Limited. All rights reserved.

The **Locate Using Value**, **Locate Using Address**, and **Array Expansion...** menu items function as described in *Watch processor view pop-up menu* on page 5-24.

Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

To display the dialog shown in Figure 5-76, select **Properties...** from the pop-up menu.

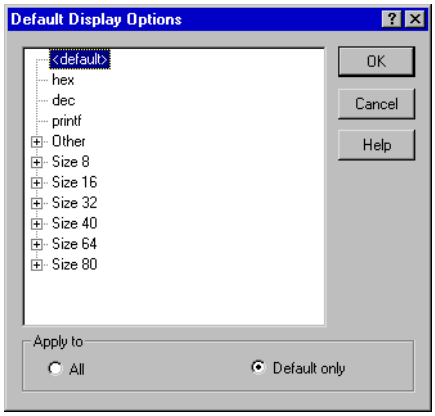



**Figure 5-76 Default Display Options dialog**

Refer to AXD online help for full details.

### 5.5.4 Breakpoints system view

The Breakpoints system view, shown in Figure 5-77, enables you to set, modify, or remove breakpoints. You can change the column widths by dragging the dividing lines between the column headings to the left or right.

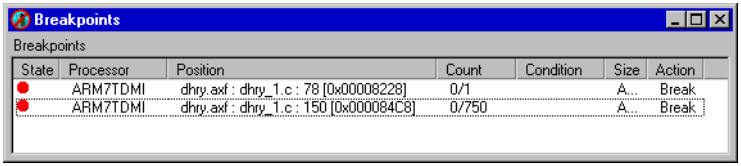



**Figure 5-77 Breakpoints system view**

You can see details of any breakpoints that are currently set. To disable an existing breakpoint, click the red disc at the left of its line. The center of the disc becomes gray. Click the disc again to restore normal operation.

To add a new breakpoint, right-click anywhere within the Breakpoints system view to display the pop-up menu shown in Figure 5-78 on page 5-59 and select **Add**.


Copyright © 1999-2001 ARM Limited. All rights reserved.

To modify a breakpoint, do either of the following:

- double-click on its line
- right-click on its line to display the pop-up menu and select **Properties**.

**Figure 5-78 Breakpoints system view pop-up menu**

The **Locate Using Address** menu item functions as described in *Watch processor view pop-up menu* on page 5-24.

Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

Whether you are adding a new breakpoint or modifying an existing breakpoint, you use the Breakpoint Properties dialog shown in Figure 5-79 on page 5-60.


Copyright © 1999-2001 ARM Limited. All rights reserved.

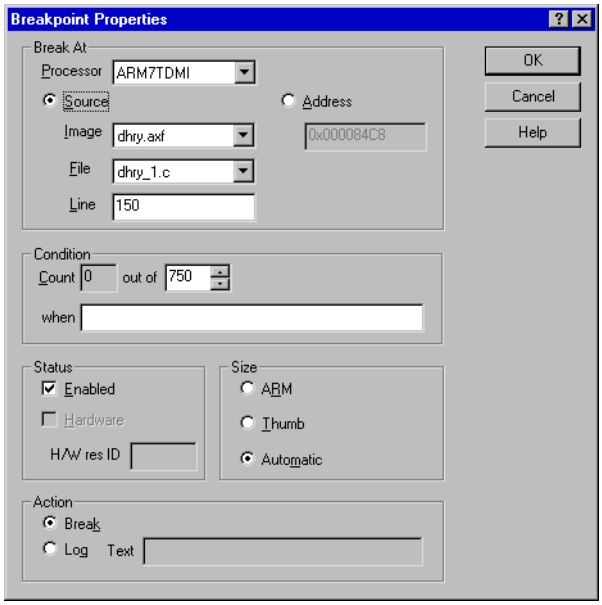



**Figure 5-79 Breakpoint Properties dialog**

The fields in the Break At group specify the location of the breakpoint. Select one processor if your target has multiple processors. You can specify a line number in a selected source file that contributes to a selected image, or you can select the **Address** radio button and specify a memory address.

The fields in the Condition group enable you to specify when arrival at the breakpoint must be ignored and when it must trigger the breakpoint. You can specify in the out of field the number of times execution must arrive at the specified location to trigger the breakpoint. Also, if you specify an expression in the when field, the count of arrivals at the breakpoint increments only if the expression evaluates to True.

**Note**

If you specify an expression that cannot be evaluated, a result of True is assumed.

Under Status, you can see whether the breakpoint is currently enabled, and change this setting if required. You can also see whether it is a software or hardware breakpoint. A hardware breakpoint can have a hardware resource identifier.

You are recommended to leave the Size set to Automatic, but you can change this to ARM (32-bit) or Thumb (16-bit) if necessary. For example, the debugger might not be able to determine whether it is debugging ARM code or Thumb code if:

- the project was built without debugging information (`-g-` switches off debugging)

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

- you are debugging a ROM image.

**Note**

In the current version of ADS it is not possible to set a bytecode-based breakpoint. If you specify a breakpoint on Jazelle instructions this creates an invalid breakpoint and might display an error message.

The setting in the Action group is normally **Break**, to stop execution when the specified conditions are met. The alternative, **Log**, adds a record in a log of events. If you select **Log**, whatever you enter in the Text field is output each time the conditions are met. To examine the log of events, select **Output** from the **System Views** menu (see *Output system view* on page 5-63). The pop-up menu of the Output system view enables you to save subsequent records in a disk file and to clear the current entries from the log.

### 5.5.5 Watchpoints system view

The Watchpoints system view, shown in Figure 5-80, enables you to set, modify, or remove watchpoints. You can change the column widths by dragging the dividing lines between the column headings to the left or right.

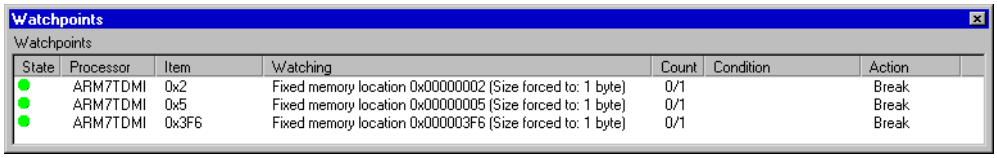



**Figure 5-80 Watchpoints system view**

You can see details of any watchpoints that are currently set. To disable an existing watchpoint, click the green disc at the left of its line. The center of the disc becomes gray. Click the disc again to restore normal operation. A disc has a red cross through it if the watchpoint is currently out of scope.

To add a new watchpoint, right-click anywhere within the Watchpoints system view to display the pop-up menu shown in Figure 5-81 on page 5-62 and select **Add**.

To modify a watchpoint, do either of the following:

- double-click on its line
- right-click on its line to display the pop-up menu and select **Properties**.


Copyright © 1999-2001 ARM Limited. All rights reserved.

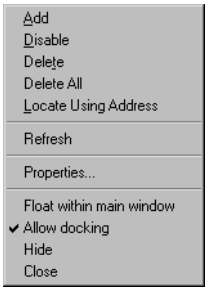



**Figure 5-81 Watchpoints system view pop-up menu**

The **Locate Using Address** menu item functions as described in *Watch processor view pop-up menu* on page 5-24.

Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

Whether you are adding a new watchpoint or modifying an existing watchpoint, you use the Watchpoint Properties dialog shown in Figure 5-82.

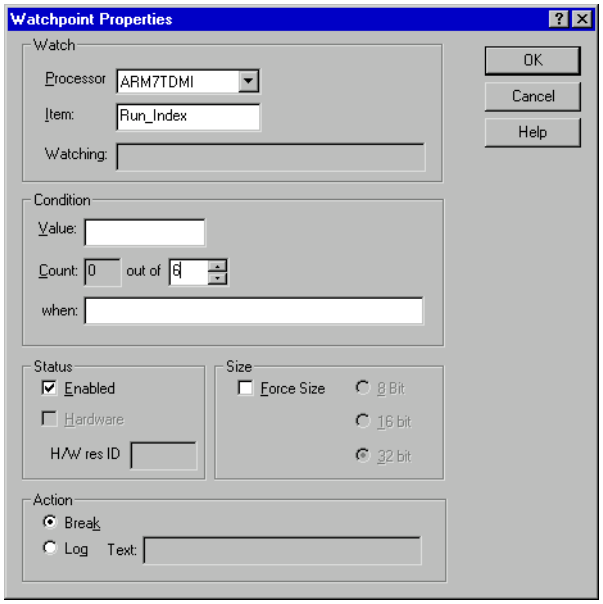



**Figure 5-82 Watchpoint Properties dialog**


Copyright © 1999-2001 ARM Limited. All rights reserved.

The fields in the Watch group specify the location of the watched value. The Processor field enables you to select one processor if your target has multiple processors. Specify in the Item field what to watch by giving the name of a variable or register, or an expression that evaluates to an address. The Watching field is read-only.

**Note**

In the current version of ADS it is not possible to set a bytecode-based watchpoint. If you specify a watchpoint on Jazelle instructions this creates an invalid watchpoint and might display an error message.

The fields in the Condition group enable you to specify when a change in the watched value must be ignored and when it must trigger the watchpoint. You can specify in the Value field a numeric constant, in which case the watchpoint is triggered only if the watched value changes to the specified value. You can specify in the out of field the number of times the watched value must change to trigger the watchpoint. Also, if you specify an expression in the when field, changes in value are counted only if the expression evaluates to True. You can concatenate conditions by using the C language && and || syntax.

**Note**

If you specify an expression that cannot be evaluated, a result of True is assumed.

Under Status, you can see whether the watchpoint is currently enabled, and change this setting if required. You can also see whether it is a software or hardware watchpoint. A hardware watchpoint can have a hardware resource identifier.

Under Size, you are recommended to leave **Force Size** unchecked. The area of memory watched is then the size of the variable if you are watching a variable, or a 4-byte word if you are watching a memory location. If you force the size of the watched area of memory you can select 8, 16, or 32 bits.

The setting in the Action group is normally **Break**, to stop execution when the specified conditions are met. The alternative, **Log**, adds a record in a log of events. If you select **Log**, whatever you enter in the Text field is output each time the conditions are met. To examine the log of events, select **Output** from the **System Views** menu (see *Output system view*). The pop-up menu of the Output system view enables you to save subsequent records in a disk file and to clear the current entries from the log.

### 5.5.6 Output system view

The Output system view enables you to examine both a list of function calls made to the *Remote Debug Interface* (RDI) and a list of log messages. These can help you determine which program statements have and have not been executed.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Select **Output** from the **System Views** menu to display a window, shown in Figure 5-83, containing two tabbed pages, labeled **RDI Log** and **Debug Log**.

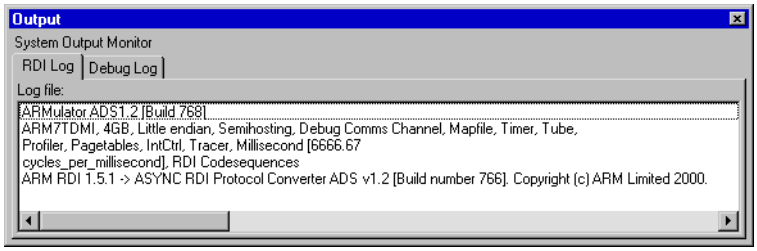



**Figure 5-83 Output system view**

Click on the **RDI Log** tab to see the page that contains a list of function calls made to the RDI. This requires the `$rdi_log` debugger internal variable to be set to 1.

Click on the **Debug Log** tab to see a list of messages recorded when execution passed through any trace points in the program (execution does not stop at an action point if you specify a trace message to be logged). The messages displayed are those specified when you defined each trace point (see *Breakpoints system view* on page 5-58). The debug log also contains any other general debugger output such as error messages.

### Output system view pop-up menu

To display the Output pop-up menu, shown in Figure 5-84, right-click on either the **RDI Log** tab or the **Debug Log** tab.

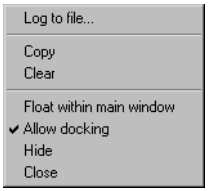



**Figure 5-84 Output system view pop-up menu**

To specify a file in which to store the lines that appear in the Output view, select **Log to file…**. You can select an existing file, or specify a new file. If you do save this information in a file, the name of the file is shown in the Output system view.

Select **Clear** to remove any lines currently displayed in the Output view.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.5.7 Command Line Interface system view

The *Command Line Interface* (CLI) system view provides you with an alternative method of issuing commands and viewing data. You enter commands in response to CLI prompts, as shown in Figure 5-85. Any data that you request is displayed in the CLI system view.

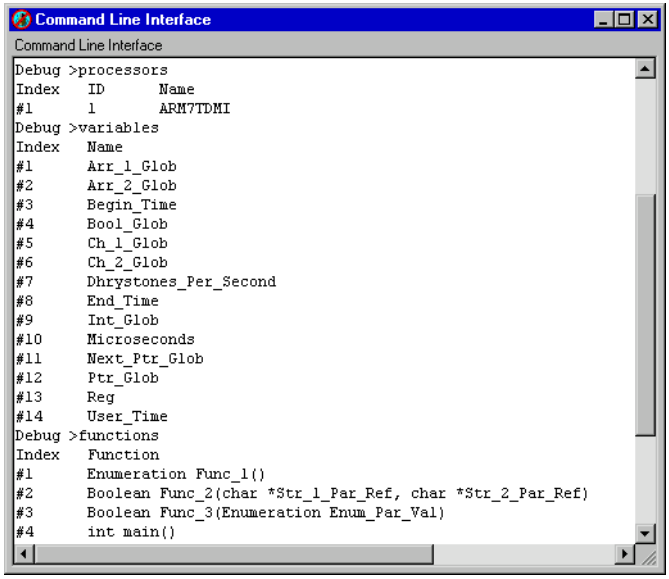



**Figure 5-85 Command Line Interface system view**

Details of all the commands you can issue and data you can display are given in Chapter 6 *AXD Command-line Interface*.

#### Command Line Interface system view pop-up menu

To display the CLI system view pop-up menu, shown in Figure 5-86 on page 5-66, right-click within the Command Line Interface system view.


Copyright © 1999-2001 ARM Limited. All rights reserved.

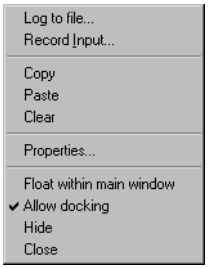



**Figure 5-86 CLI system view pop-up menu**

**Log to file…** enables you to start or stop recording in a disk file everything that appears in the CLI system view.

**Record Input**… enables you to start or stop recording in a disk file every command that you enter in the CLI system view.

**Clear** enables you to clear the current contents of the CLI system view.

Refer to AXD online help for details of all the pop-up menu items.

Select **Properties...** from the pop-up menu to change the CLI system view properties. This displays the CLI Properties dialog which contains three tabbed pages entitled:

- **General**, shown in Figure 5-87 on page 5-67
- **Format**, shown in Figure 5-88 on page 5-68
- **Files**, shown in Figure 5-89 on page 5-68.

When you have made changes on any of the tabbed pages, click:

**OK** To accept all the current settings on all the tabbed pages, and close the dialog.

**Cancel** To ignore any changes made since the dialog was opened or since the **Apply** button was last clicked, and close the dialog.

**Apply** To accept all the current settings on all the tabbed pages, and leave the dialog open for further changes.

**Help** To display relevant online help.

Figure 5-87 on page 5-67 shows the CLI Properties dialog with the default **General** tab selected.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

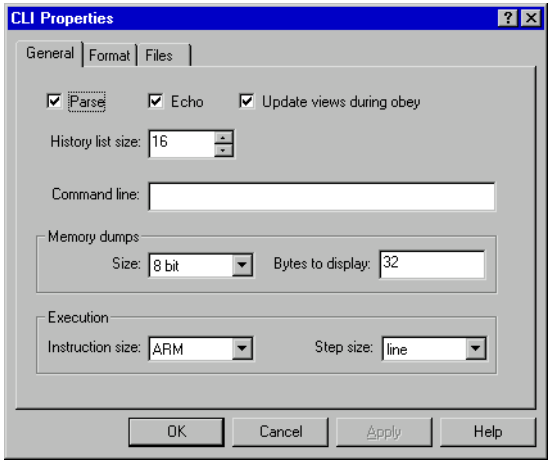



**Figure 5-87 Command Line Interface Properties dialog, General tab**

Leave **Parse** checked. Your CLI commands are then validated when you enter them and translated into internal commands.

The **Echo** setting specifies whether commands read from a file by an Obey command are displayed in the CLI system view. This also determines whether they are logged.

The **Update views during obey** setting enables you to control whether or not screen updates take place while commands are being executed from an Obey file. If you have several open views, they are all normally updated every time the script causes a break in execution, slowing down AXD significantly. Clear this check box to allow the script to run and update the screen just once, when it terminates. The setting persists, with other CLI properties. If a script modifies this CLI property, it is reset to its original state when the script terminates.

The number in the History list size field sets the number of CLI commands that you can recall using the up and down arrow keys or the Ctrl+PageUp key combination. To examine recent commands, or to use a recent command as the basis for a new command, see *Command history* on page 6-3.

Click on the **Format** tab to display the dialog shown in Figure 5-88 on page 5-68.


*Copyright © 1999-2001 ARM Limited. All rights reserved.*

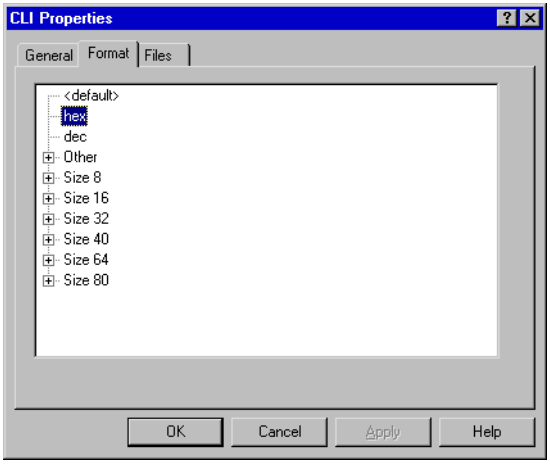



**Figure 5-88 Command Line Interface Properties dialog, Format tab**

Use this dialog to set the default format for displaying data. This defines the appearance of values displayed in response to such commands as Memory.

Click on the + sign of the data size you want to be displayed. A further list shows you all the formats valid for that size and enables you to choose one.

Click on the **Files** tab to display the dialog shown in Figure 5-89.

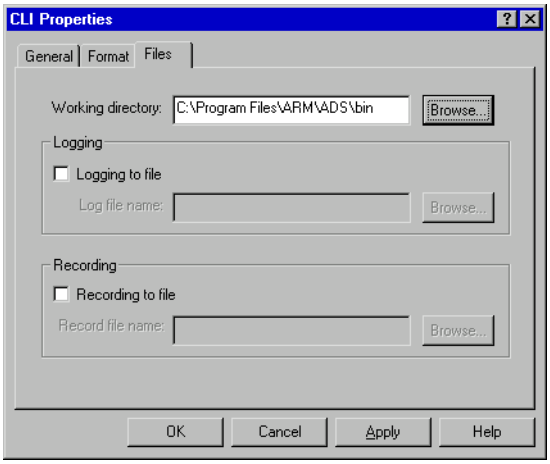



**Figure 5-89 Command Line Interface Properties dialog, Files tab**

AXD online help gives details of all the fields in the tabbed pages of the CLI Properties dialog.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.5.8 Debugger Internals system view

The Debugger Internals system view has two tabbed pages:

- *Internal Variables*
- *Statistics* on page 5-72.

#### Internal Variables

The first tabbed page of the Debugger Internals system view shows **Internal Variables**, as shown in Figure 5-90.

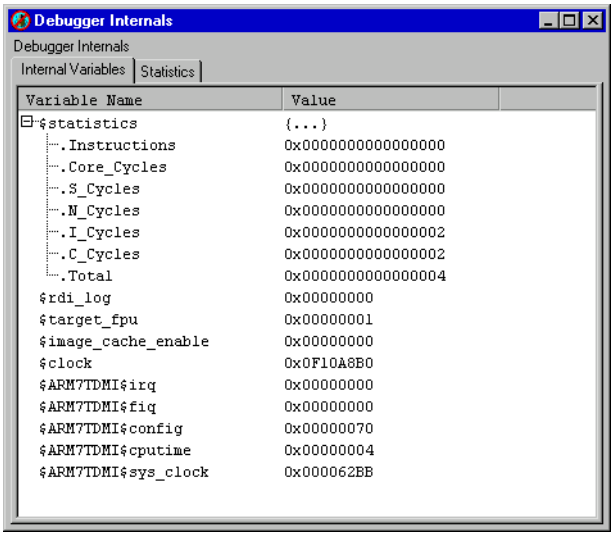



**Figure 5-90 Debugger Internals, Internal Variables**

The debugger, like most programs, uses variables. The internal variables used by the debugger depend on the target in use. If you are using Multi-ICE to debug a hardware target, for example, you will see different internal variables from those described here. If you are using ARMulator, the variables displayed depend on the processor you are simulating. They generally include the following:

**$statistics** This is a group of internal variables that you can examine more clearly on the **Statistics** tab (see *Statistics* on page 5-72) or by using a CLI command (see *statistics* on page 6-54).

**$rdi_log** This variable controls how target information is logged in the **RDI Log** tab of the Output system view, shown in Figure 5-83 on page 5-64. If it is unset (the default) then no logging occurs. The two least significant bits have the following meanings:

**Bit 0** RDI (0 = off, 1 = on).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

**Bit 1** Device Driver Logging (0 = off, 1 = on).

This variable is used for diagnostic purposes to track communication between the debugger and the target and so is not normally required.

**$target_fpu** This is an enumeration that controls the way that floating point numbers are displayed by the debugger. It is important to ensure the correct display of float and double values in memory that this variable is set to a value that is appropriate for the target in use.

If you attempt to change this value, a validity test checks that the new settings are compatible with the representation of floating point values in the current image. Valid settings and their meanings are:

**1** Selects pure-endian doubles (`softVFP`). This is the default setting for images built with ADS tools. Values are read from ordinary registers.

**2** Selects mixed-endian doubles (`softFPA`). Values are read from ordinary registers.

**3** Selects hardware Vector Floating Point unit (`VFP`). Values are read from registers CP10 and CP11.

**4** Selects hardware Floating Point Accelerator (`FPA`). Values are read from registers CP1 and CP2.

**5** Reserved.

Incompatible settings are accepted by the debugger but a warning is given.

SoftVFP and SoftFPA images run correctly on a target whether or not hardware floating point is present. FPA images can also run correctly without hardware floating point, but only if the Floating Point Emulator in ARMulator is active. VFP images require appropriate hardware, or an ARMulator that simulates it.

For further details, and details of the software to install appropriate support code, see the *ADS Compilers and Libraries Guide*.

**$image_cache_enable**

This variable holds internal debugging information when using *Trace Debug Tools* (TDT) on a target that must not stop execution. Such information would otherwise be lost and so is held locally in the host computer memory. This information is useful only to the debugger and cannot be accessed directly.

**$clock** This variable applies to ARMulator only and is based on the ARMulator clock speed setting. This variable is unavailable where the ARMulator clock speed is set to real time. Where the ARMulator clock speed is set to simulated, this variable contains the number of simulated

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

microseconds that have elapsed since the application program began execution (see *Configure Target...* on page 5-87). This variable is read-only.

In addition to these variables, some debug targets can create their own variables. These are named `$<proc_name>$<var_name>`, where:

`<proc_name>` is the name of the processor, as shown in the **Target** tab of the Control system view (for example, `ARM720T`).

`<var_name>` is the name of the variable, and can include:

- `irq` (For example, `$ARM720T$irq`.) A target can export this variable to provide a means of asserting the interrupt request pin. To trigger an interrupt manually, set the value to 1. To clear the interrupt, set the value to 0. To take the interrupt exception a processor must have IRQ enabled in the CPSR.
- `fiq` (For example, `$ARM720T$fiq`.) A target can export this variable to provide a means of asserting the fast interrupt request pin. To trigger a fast interrupt manually, set the value to 1. To clear the fast interrupt, set it to 0. To take the interrupt exception a processor must have FIQ enabled in the CPSR.
- `cputime` (For example, `$ARM720T$cputime`.) This variable applies to ARMulator only and contains the best estimate of the time the processor has been running, measured in clock units. A clock unit is the reciprocal of the ARMulator clock speed setting. This variable is unavailable where the ARMulator clock speed is set to real time (see *Configure Target...* on page 5-87). This variable is read only.

Your debug target might create other variables. See the target documentation for details.

You can examine the contents of all these variables, and change the values stored in some of them. For more information about data display formats and data entry formats, see *Data formatting* on page 4-16.

### Debugger Internal Variables pop-up menu

Right-click inside the Debugger Internals system view with the **Internal Variables** tab selected to display the pop-up menu shown in Figure 5-91 on page 5-72.


Copyright © 1999-2001 ARM Limited. All rights reserved.

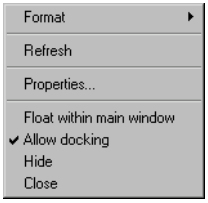


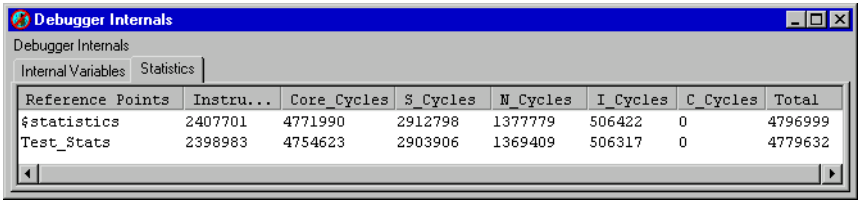

**Figure 5-91 Internal Variables pop-up menu**

Use this pop-up menu to set properties and to select a display format. Refer to AXD online help for details.

Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.

## Statistics

The second tabbed page of the Debugger Internals system view is available only when you use a target simulated by software. The page shows statistics, as in Figure 5-92.




| Reference Points | Instru... | Core_Cycles | S_Cycles | N_Cycles | I_Cycles | C_Cycles | Total |
|---|---|---|---|---|---|---|---|
| $statistics | 2407701 | 4771990 | 2912798 | 1377779 | 506422 | 0 | 4796999 |
| Test_Stats | 2398983 | 4754623 | 2903906 | 1369409 | 506317 | 0 | 4779632 |

**Figure 5-92 Debugger Internals, von Neumann core statistics**

A group of debugger internal variables contains statistics relating to your current debugging session. These variables are displayed more clearly on the **Statistics** tab than on the **Internal Variables** tab. Drag the column divider lines to the left or right to change the column widths if necessary.

The first line of statistics shows values accumulated from the beginning of execution of the program you are debugging, and is labeled `$statistics` (see also the CLI command *statistics* on page 6-54).

You can add more lines of statistics, accumulated from later interruptions of program execution. When execution has stopped, to start accumulating a new line of statistics, right-click in the **Statistics** tab of the Debugger Internals system view, and select **Add New Reference Point**.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Your debug target might display other statistics. See the *ADS Debug Target Guide* for full information on the different cycle types that might be displayed and their meaning. Two examples are shown here:

- *Statistics for von Neumann debug targets*
- *Statistics for Harvard debug targets* on page 5-74.

### Statistics for von Neumann debug targets

When simulating von Neumann architecture cores such as the ARM7TDMI core, shown in Figure 5-92 on page 5-72, the following information is displayed:

**Reference Points**
The name you specify to identify each line of statistics that you add.

**Instructions** The number of program instructions executed.

**Core_Cycles**
Internal core cycles indicating the time an instruction spends in the execute stage of the pipeline.

**S_Cycles** The number of sequential cycles performed. The CPU requests transfer to or from the same address, or an address that is a word or halfword after the preceding address.

**N_Cycles** The number of nonsequential cycles performed. The CPU requests transfer to or from an address that is unrelated to the address used in the preceding cycle.

**I_Cycles** The number of internal cycles performed. The CPU does not require a transfer because it is performing an internal function (or running from cache).

**C_Cycles** The number of coprocessor cycles performed.

**Total** The sum of the S_Cycles, N_Cycles, I_Cycles, and C_Cycles.

If you use a map file (see *ARMulator configuration* on page 5-88) the display shows additional information, including:

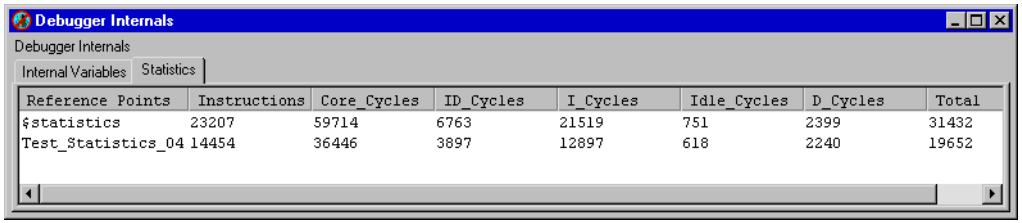
**Wait_States** The number of wait-states added by the Mapfile component.

**True_Idle_Cycles**
The number of I_Cycles less the number that are part of an I-S pair. It is only displayed if you set SpotISCyles to True

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Statistics for Harvard debug targets

When simulating Harvard architecture cores such as the ARM9 core and StrongARM®, different statistics are accumulated, shown in Figure 5-93.




| Reference Points | Instructions | Core_Cycles | ID_Cycles | I_Cycles | Idle_Cycles | D_Cycles | Total |
|---|---|---|---|---|---|---|---|
| $statistics | 23207 | 59714 | 6763 | 21519 | 751 | 2399 | 31432 |
| Test_Statistics_04 | 14454 | 36446 | 3897 | 12897 | 618 | 2240 | 19652 |

**Figure 5-93 Debugger Internals, Harvard core statistics**

In these cases, the meanings are:

**Reference Points**
The name you specify to identify each line of statistics that you add.

**Instructions** The number of program instructions executed.

**Core_Cycles**
The total number of core clock ticks, including stalls due to interlocks and instructions that take more than one cycle.

**ID_Cycles** Cycles in which both the instruction bus and the data bus were active.

**I_Cycles** Cycles in which the instruction bus was active and the data bus was idle.

**Idle_Cycles** Cycles in which both the instruction bus and the data bus were idle.

**D_Cycles** Cycles in which the data bus was active and the instruction bus was idle.

**Total** The sum of cycles on the memory bus.

### Statistics pop-up menu

Right-click inside the Debugger Internals system view with the **Statistics** tab selected to display the pop-up menu shown in Figure 5-94 on page 5-75.


Copyright © 1999-2001 ARM Limited. All rights reserved.

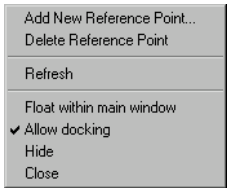



**Figure 5-94 Statistics pop-up menu**

Use this pop-up menu to add a new line of statistics to the displayed table, or to delete the currently selected line. Refer to AXD online help for details.

Select **Refresh** to update and recalculate the displayed data values. This item is useful if the target supports RealMonitor. See also *Refresh All* on page 5-100.


*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 5.6 Execute menu

The **Execute** menu (see Figure 5-95), lets you control how execution continues from the current point.

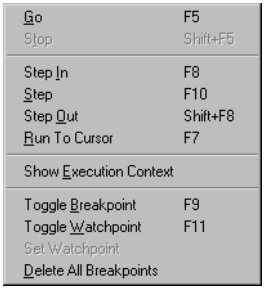



**Figure 5-95 Execute menu**

The **Execute** menu items are described under the following headings:

- *Go*
- *Stop* on page 5-77
- *Step In* on page 5-77
- *Step* on page 5-77
- *Step Out* on page 5-78
- *Run To Cursor* on page 5-78
- *Show Execution Context* on page 5-78
- *Toggle Breakpoint* on page 5-78
- *Toggle Watchpoint* on page 5-79
- *Set Watchpoint* on page 5-79
- *Delete All Breakpoints* on page 5-79.

### 5.6.1 Go

This begins execution. If you have loaded an image but not yet run it, execution starts from the first executable instruction. If execution is currently stopped, at a breakpoint for example, then it resumes from the point at which it stopped.

When you start executing an image, AXD tries to locate the relevant source files. If they are not found, you are asked to specify their location in the dialog shown in Figure 5-96 on page 5-77.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

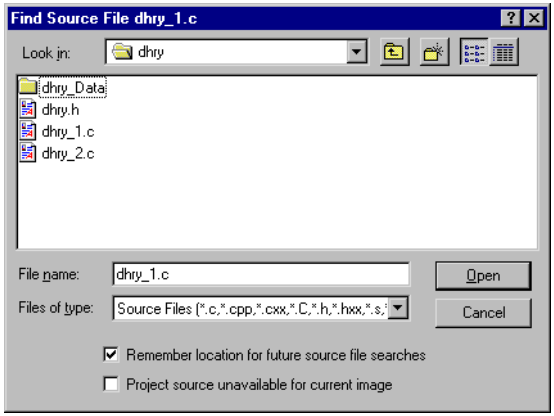



**Figure 5-96 Find Source dialog**

If you select the check box **Remember location for future source file searches**, AXD finds these files without your help in subsequent sessions.

If you select the check box **Project source unavailable for current image**, AXD continues the session without access to any source files.

### 5.6.2 Stop

This menu item is enabled only when the program is executing. It stops execution as soon as the program can be interrupted.

### 5.6.3 Step In

This executes the current instruction and stops. If the current instruction is a call to a function, then it stops at the first executable instruction in that function.

This menu item is not enabled when executing Jazelle instructions as stepping options are not available for these instructions.

In addition, if the current instruction is a BXJ instruction this menu item is not enabled.

### 5.6.4 Step

This executes the current instruction and stops. If the current instruction is a call to a function, then it executes the function and stops when control returns to the caller.


Copyright © 1999-2001 ARM Limited. All rights reserved.

A C++ program might contain many calls to library functions that the compiler replaces with inline code if you choose to compile for high speed rather than small size. This prevents the Step command from behaving as expected. A C++ compiler option is available to force calls to library functions to be compiled as calls in such cases. For further information refer to the *ADS Compilers and Libraries Guide*.

This menu item is not enabled when executing Jazelle instructions as stepping options are not available for these instructions.

In addition, if the current instruction is a BXJ instruction this menu item is not enabled.

### 5.6.5 Step Out

This completes execution of the current function and stops when control returns to the caller.

This menu item is not enabled when executing Jazelle instructions as stepping options are not available for these instructions.

### 5.6.6 Run To Cursor

This continues execution but stops when the next instruction to be executed is the one where you have positioned the cursor.

**Note**

As **Run to Cursor** is dependent on setting a breakpoint, it is not possible to select this option for Jazelle instructions where breakpoints are currently not supported.

### 5.6.7 Show Execution Context

This selects **Show Execution Context** when you are viewing either the source code or the disassembled code related to a halted process. The area of code displayed changes so that the visible lines of code are replaced by the lines surrounding the current execution position.

### 5.6.8 Toggle Breakpoint

When you are viewing a source file or a disassembly, you can set or remove a breakpoint at the current cursor position by selecting **Toggle Breakpoint** from the **Execute** menu.

You can also set or remove a breakpoint by double-clicking in the margin of the required line in a source or disassembly view, or by right-clicking on the line and selecting **Toggle Breakpoint** from the pop-up menu.


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Note**

In the current version of ADS it is not possible to set a bytecode-based breakpoint. If you specify a breakpoint on Jazelle instructions this creates an invalid breakpoint and might display an error message.

### 5.6.9 Toggle Watchpoint

When you are viewing a disassembly, you can set or remove a watchpoint on the currently selected item by selecting **Toggle Watchpoint** from the **Execute** menu.

**Note**

In the current version of ADS it is not possible to set a bytecode-based watchpoint. If you specify a watchpoint on Jazelle instructions this creates an invalid watchpoint and might display an error message.

### 5.6.10 Set Watchpoint

When you are viewing a source file, you can set or replace a watchpoint on the currently selected item by selecting **Set Watchpoint** from the **Execute** menu.

**Note**

In the current version of ADS it is not possible to set a bytecode-based watchpoint. If you specify a watchpoint on Jazelle instructions this creates an invalid watchpoint and might display an error message.

### 5.6.11 Delete All Breakpoints

To delete all currently set breakpoints, select **Delete All Breakpoints** from the **Execute** menu.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 5.7 Options menu

The **Options** menu, shown in Figure 5-97, enables you to examine and change a variety of settings, including some that affect the appearance of the debugger screen. This menu also enables you to start and stop profiling.

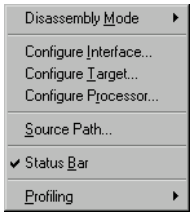



**Figure 5-97 Options menu**

The **Options** menu items are described under the following headings:

- *Disassembly Mode*
- *Configure Interface...*
- *Configure Target...* on page 5-87
- *Configure Processor...* on page 5-96
- *Source Path...* on page 5-98
- *Status Bar display control* on page 5-98
- *Profiling* on page 5-98.

### 5.7.1 Disassembly Mode

This applies only when you have a Disassembly processor view selected. To specify the type of disassembly you require, select **Disassembly Mode** from the **Options** menu. A submenu appears, enabling you to select ARM/Thumb Mixed, ARM, Thumb or ByteCode. One of these is checked, indicating the current disassembly mode.

In ARM/Thumb Mixed mode, the debugger uses information read while loading the image to set the appropriate mode. This is possible only when debugging information is present, so cannot be done if, for example, the image is in ROM. The default setting then used might not always be correct.

### 5.7.2 Configure Interface...

To configure the AXD user interface, select **Configure Interface...** from the **Options** menu. The resulting dialog has tabbed pages entitled:

- General
- Views

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

- Formatting
- Session File
- Toolbars
- Timed Refresh.

To display detailed information about the features of the currently displayed tabbed page, click **Help**.

When you have made changes on one or more of these tabbed pages, you can apply or abandon the changes as follows:

**OK** Apply outstanding changes on all tabbed pages and close the dialog.

**Cancel** Ignore any outstanding changes and close the dialog.

**Apply** Apply outstanding changes on all tabbed pages and keep the dialog open.

### General

Figure 5-98 shows the **General** tab of the Configure Interface dialog.

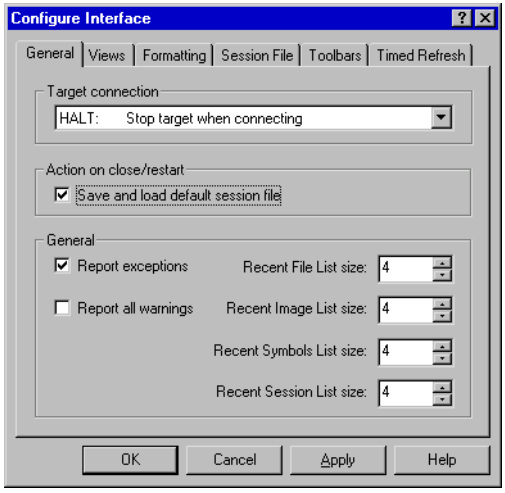



**Figure 5-98 Configure Interface, General tab**

The **General** tab of the Configure Interface dialog enables you to control the behavior of the target processor when you connect the debugger to it, and the actions to be taken when you restart or close a debugging session. It also enables you to make some other general settings applicable to the whole debugging session.

Most targets stop execution when a debugger is connected and restart when instructed to do so. Also, the displayed views are all updated each time the target execution stops. All views therefore show consistent data at all times.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Targets that support RealMonitor can be traced and queried without interrupting execution. The Target connection drop-down list enables you to select an appropriate way of connecting the debugger to the target, depending on whether you are using RealMonitor. See *RealMonitor support* on page 4-14 for more information.

**Halt** The debugger is allowed to, and does, stop execution of the target when the connection is made. If connection to a new target is requested during program execution a warning message is displayed so that execution on the current target can continue, or stop, as required. This is the default setting.

**NoHalt** The target is assumed to be executing and must not be interrupted. If the target supports *RealMonitor* and a non-intrusive connection can be made, then the connection is made. Otherwise the debugger redisplays the configuration dialog.

**Attach** If the target supports *RealMonitor* and a non-intrusive connection can be made, then the connection is made without stopping target execution. If the target does not support *RealMonitor* then the connection is still made even though doing so stops the target execution.

**Note**

The AXD debug architecture does not currently support attaching and re-attaching while using any kind of semihosting.

Under Action on close/restart you can select a **Save and load session file** check box. If this is checked, details of your debug session are saved in a session file when you end the session, and next time you run AXD the new session starts in the same state. If the **Save and load session file** check box is cleared, details are not saved at the end of the current session, and the next session begins in the usual default state.

The check boxes in the General group control the types of messages recorded in the **Debug Log** tabbed page of the Output system view (see *Output system view* on page 5-63), and the List size fields allow you to control the amount of recent history that is maintained.

### Views

Figure 5-99 on page 5-83 shows the **Views** tab of the Configure Interface dialog.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

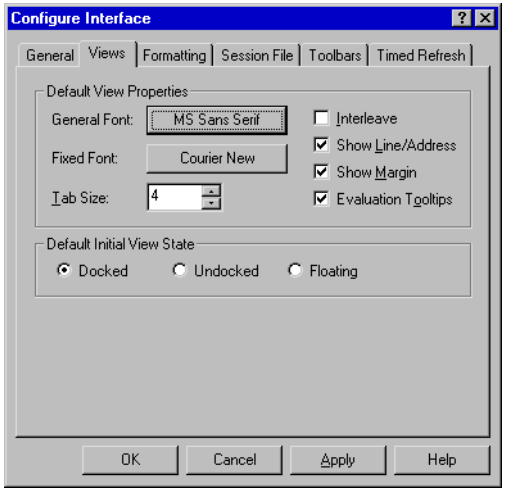



**Figure 5-99 Configure Interface, Views tab**

These Default view properties are used as default settings in all displayed views.

The General Font you select applies to the following views:

- Backtrace
- Breakpoints
- Control Monitor
- Low Level Symbols
- Output
- Watchpoints.

The Fixed Font you select applies to the following views:

- Command Line Interface
- Comms Channel
- Console
- Debugger Internals
- Disassembly
- Memory
- Registers
- Source
- Variables.


Copyright © 1999-2001 ARM Limited. All rights reserved.

The Default Initial View State you select applies to the starting state of all views except Source and Disassembly views which always float within the main window. You can set a view window as:

- Docked
- Undocked
- Floating.

Docked and floating windows are described in *Docked and floating windows* on page 2-10. See also *Window menu* on page 5-99.

To display detailed information on all the fields and check boxes on this tabbed page, click **Help**.

### Formatting

Figure 5-100 shows the **Formatting** tab of the Configure Interface dialog.

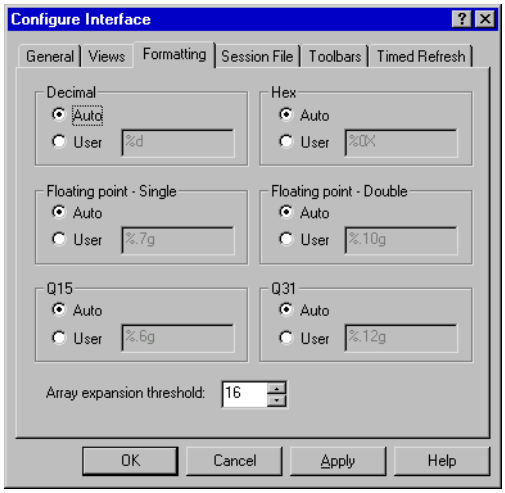



**Figure 5-100 Configure Interface, Formatting tab**

The **Formatting** tab of the Configure Interface dialog enables you to define the formatting strings used for the default formatting options decimal, hex, floating point single, floating point double, Q15, and Q31. To change the default formatting string for a format option, select **User**.

The value you set for the Array expansion threshold limits the number of child items that you can display without first displaying the array expansion pop-up.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Session File

Figure 5-101 shows the **Session File** tab of the Configure Interface dialog. This tabbed page enables you to make settings that apply to all session files that you might create, including the default session file created automatically at the end of each debug session.

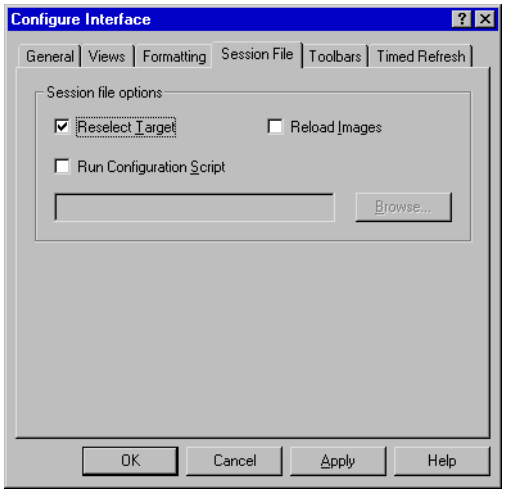



**Figure 5-101 Configure Interface, Session File tab**

Under Session file options, you can choose whether or not to **Reselect Target**. If this is checked, target details are saved at the end of a session and a new session connects to the same target as the previous session. If unchecked, the new session starts with the same settings and displayed views as the previous session but with no target selected.

If you do reselect the previous target, you can use **Reload Images** to choose whether or not to reload the previous image onto the target. If this is checked, image details are saved at the end of a session and a new session loads the previous image. If unchecked, the new session starts with the same settings and displayed views as the previous session but with no image loaded.

If you use the **Browse** button to locate and select a script file, and check the **Run Configuration Script** check box, then the commands in the specified script file are executed after loading the session file and connecting to the target, but before loading any images.

### Toolbars

Figure 5-102 on page 5-86 shows the **Toolbars** tab of the Configure Interface dialog.


Copyright © 1999-2001 ARM Limited. All rights reserved.

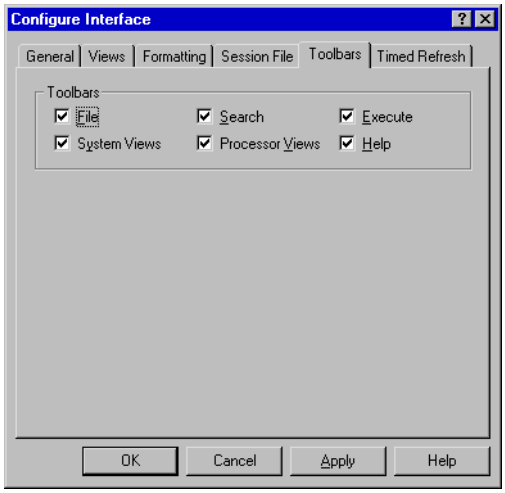



**Figure 5-102 Configure Interface, Toolbars tab**

The check boxes in the Toolbars group control the display of the named toolbars. When a toolbar name is checked in this dialog, that toolbar is displayed on the main AXD screen. These toolbars are shown in *Toolbars* on page 5-3.

### Timed Refresh

Figure 5-103 shows the **Timed Refresh** tab of the Configure Interface dialog.

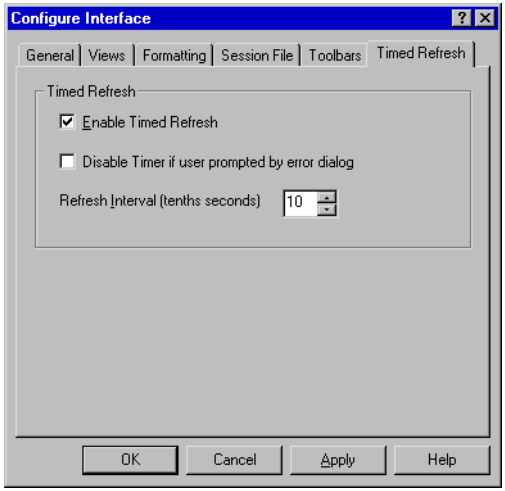



**Figure 5-103 Configure Interface, Timed Refresh tab**


Copyright © 1999-2001 ARM Limited. All rights reserved.

The **Timed Refresh** tab of the Configure Interface dialog is particularly useful when you are debugging a target that supports *RealMonitor* (see *RealMonitor support* on page 4-14).

When you debug a target that does not support *RealMonitor*, all displayed views are refreshed each time execution on the target stops. This means that all the information you can see is consistent.

Execution on a target that supports *RealMonitor*, however, can be continuous, with each displayed view showing information that was relevant at one time but not necessarily at the time that the information in any other view was captured.

The pop-up menu available in most views includes a **Refresh** item, but that refreshes the information in that view only. The **Window** menu includes a **Refresh All** item, to refresh all the displayed views at the same time.

If you select the **Enable Timed Refresh** check box on the **Timed Refresh** tab of the Configure Interface dialog so that it is checked, then all displayed views are refreshed automatically and regularly.

Timed refresh is automatically suspended when dialogs are displayed. You can check the **Disable Timer if user prompted by error dialog** check box on the **Timed Refresh** tab of the Configure Interface dialog so that refreshes are disabled if an error dialog is displayed. When you have cleared the error dialog, you can enable timed refresh by clicking on the **Timed Refresh** tool.

You can also check or clear the **Enable Timed Refresh** check box by selecting the **Timed Refresh** item on the **Window** menu or by clicking on the **Timed Refresh** tool.

### 5.7.3 Configure Target...

You can select and configure a debug target when you start up AXD (see *Starting and closing AXD* on page 2-3). The **Configure Target...** item on the **Options** menu enables you to change the debug target and its configuration during a debug session.

First, a Choose Target dialog displays a list of available targets, as shown in Figure 5-104 on page 5-88.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

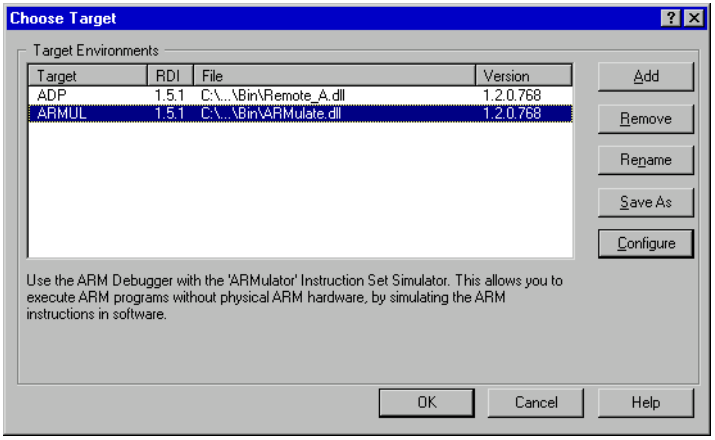



**Figure 5-104 Choose Target dialog**

If the target you want is not in the list, click the **Add** button to locate and select the required `.dll` file. The selection list shows all available `.dll` files including, for example `etm.dll` which is only fully functional if you are licensed to use the *Trace Debug Tools* (TDT) add-on product.

When you can see the target you want in the list, select that line as shown in Figure 5-104, and click the **Configure** button.

The appearance of the configuration dialog depends on the target you selected. Examples follow showing:

- *ARMulator configuration*
- *Multi-ICE configuration* on page 5-93
- *Remote_A configuration* on page 5-94

**Note**

In some of these procedures you need to locate and select a required file. A browse dialog helps you do this. However, files of the type you require might not be listed unless you select **Windows Explorer** → **View** → **Options...** → **Show all files**.

### ARMulator configuration

If you need to add ARMulator to the list of available targets in the Choose Target dialog, click **Add** and in the resulting browse dialog locate and select the `armulate.dll` file.

Select the ARMulator target line and click the **Configure** button to display the dialog shown in Figure 5-105 on page 5-89.


Copyright © 1999-2001 ARM Limited. All rights reserved.

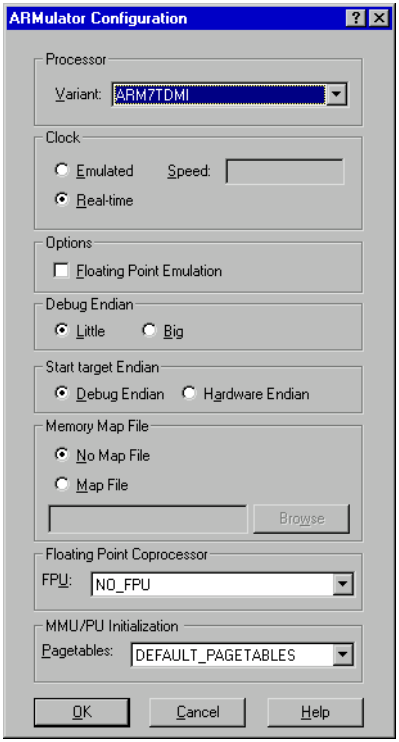



**Figure 5-105 ARMulator Configuration dialog**

The ARMulator Configuration dialog enables you to examine and change the following settings:

**Processor** Use the drop-down list to specify which ARM processor you want ARMulator to simulate.

The list of processors includes all available variants including, for example `ARM7TDMI-ETM` or `ARM920T-ETM`. ARMulator can simulate these Embedded Trace Macrocells but full trace functionality is only available if you are licensed to use the *Trace Debug Tools* (TDT) add-on product.

**Clock** Choose between simulating a processor clock running at a speed that you can specify, or executing instructions in real time. Entering a speed without specifying units assumes Hz, for example `50` assumes `50Hz`. Speeds given in kHz and GHz are also acceptable.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

**Options** The Floating Point Emulator (FPE) emulates the Floating Point Accelerator (FPA) coprocessor and enables the execution of floating point instructions not supported by the main processor. Check this option to enable FPE.

**Debug Endian**

Select the byte order of the target system. This setting:

- Sets the debugger to work with the appropriate byte order.
- Sets the byte order of ARMulator models that do not have a CP15 coprocessor.
- Sets the byte order of ARMulator models that do have a CP15 coprocessor if the Start target Endian option is set to **Debug Endian**.

For further information see the summary of Endian settings in Table 5-1 on page 5-91.

**Start target Endian**

Select the way in which the byte order of ARMulator models that have a CP15 coprocessor is determined:

- Select the **Debug Endian** radio button to instruct the model to use the byte order set in the Debug Endian group.
- Select the **Hardware Endian** radio button to instruct the model to simulate the behavior of real hardware. On reset, the core model starts in little-endian mode. If the rest of the system is big-endian, you must set the big-endian bit in CP15 in your initialization code to change the core model to big-endian mode.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

You can set various combinations of the radio buttons in the Debug Endian and Start target Endian groups. Use the possible combinations as shown in Table 5-1.

**Table 5-1 Endian settings**

| Debug Endian | Start target Endian | Usage |
|---|---|---|
| Little | Debug Endian | Use this for a core or system that is always little-endian only (for example, BigEnd pin = 0). This is the default. |
| Big | Debug Endian | Use this for a core or system that is always big-endian only (for example, BigEnd pin=1). |
| Big | Hardware Endian | Use this for a big-endian system, where the core starts up in little-endian mode, but which is switched to big-endian by writing to CP15 in the initialization code. (The initialization code must be written in an endianness-independent way, that is word accesses only.) |
| Little | Hardware Endian | This combination is not required at present. |

### Memory Map File

Specify a memory map file, or that you want to use default settings.

### Floating Point Coprocessor

Where you are using a floating point coprocessor (FPA), use the drop-down list to specify the variant supported by ARMulator, for example `ARM926EJ-S` with *Vector Floating Point* (`VFP`). The default is `No_FPU`.

### MMU/PU Initialization

Specify the initialization of the *Memory Management Unit* (MMU) or *Protection Unit* (PU) for your target processor. If you are using the ARMulator to simulate a processor with an MMU, `DEFAULT_PAGETABLES` is the required setting. For PU processors, or processors where the MMU is disabled, select `NO_PAGETABLES`.

When developing your own pagetable initialization software in the ARMulator you might want to disable the MMU by selecting `NO_PAGETABLES`. This means that settings in the `peripherals.ami` configuration file are ignored.


Copyright © 1999-2001 ARM Limited. All rights reserved.

See *ARM Architecture Reference Manual* for full information on MMU/PU operation.

See *ADS Debug Target Guide* for full information on pagetables.

When you are changing settings in the ARMulator Configuration dialog you should remember the following:

- If you are using the software floating point C libraries, ensure that the **Floating Point Emulation** option is **off** (blank), its default setting. Turn the option **on** (checked) only if you want *Floating Point Emulation* (FPE) software to be loaded into ARMulator so that you can execute code that uses the *Floating Point Accelerator* (FPA) instruction set.
- Changes to the ARMulator Configuration dialog do not affect the `$target_fpu` debugger internal.
- If, in the Memory Map File group, you select **No Map File**, the memory model declared as default in the `default.ami` file is used. This typically represents a flat 4GB bank of ideal 32-bit memory having no wait states. To use a memory map file, select **Map File**. Specify the filename (for example, `armsd.map`) by entering it, or click the **Browse** button, locate and select the file, and click **Open**. You must specify an existing memory map file. For more information about ARMulator and memory map files, see the *ADS Debug Target Guide*.

When you are setting options in the Clock group you should remember the following:

- If you set a nonzero simulated Clock Speed, then the clock speed used is the value that you enter. Clock speeds can be entered in units of Hz, kHz or GHz. If you do not specify the units then Hz is assumed. Values stored in the debugger internal variable `$clock` depend on this setting, and are unavailable if you select **Real-time**. For information about debugger internal variables, see *Debugger Internals system view* on page 5-69.
- The AXD clock speed defaults to real time for compatibility with the defaults of armsd. Selecting **Real-time** in AXD is equivalent to omitting the `-clock` armsd option on the command line. In other words, the clock frequency is unspecified, and the default clock frequency specified in the configuration file `default.ami` is used (`DEFAULT_CPUSPEED=20MHz`).
- For ARMulator, you do not have to specify a clock frequency because ARMulator does not use it to simulate the execution of instructions and count cycles (for `$statistics`). However, your application program might sometimes need to

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

access a clock, so ARMulator must always be able to give clock information. ARMulator uses the clock frequency from the configuration file if you do not specify a simulated clock speed.

— In either case, ARMulator uses the clock information to calculate the elapsed time since execution of the application program began. This elapsed time can be read by the application program using the C function `clock()` or the semihosting `SYS_CLOCK`, and is also visible to the user from the debugger as `$clock`. It is also used internally by ARMulator in the calculation of `$memstats`. The clock speed (whether specified or unspecified) has no effect on actual (real time) speed of execution under ARMulator. It affects the simulated elapsed time only.

— `$memstats` is handled slightly differently because it does require a defined clock frequency so that ARMulator can calculate how many wait states are required for the memory speed defined in an `armsd.map` file. If you specify a clock speed and an `armsd.map` file is present, then `$memstats` gives useful information about memory accesses and times. Otherwise, for calculating the wait states, a default core:memory clock ratio specified in the configuration file is used, so that `$memstats` can still give useful memory timings.

See the *ADS Debug Target Guide* for full information on ARMulator configuration settings and configuration files.

### Multi-ICE configuration

If you need to add Multi-ICE to the list of available targets, click **Add** and use the resulting browse dialog to locate and select the `Multi-ICE.dll` file.

Select the Multi-ICE target line and click the **Configure** button to display the Multi-ICE configuration dialog.

The settings available in this dialog include:

- the network address of the computer running the Multi-ICE Server software
- the selection of a processor driver
- a connection name (required only when access to the Multi-ICE Server software is across a network).

Some versions of Multi-ICE might also allow you to select a `.dll` file to use as a Debug Communications Channel (DCC) viewer. Do not enable any DCC viewer from this dialog. Instead, use the AXD built-in viewer available from the **Processor Views** menu and enabled from the Processor Properties dialog. For more details see *Comms Channel processor view* on page 5-37 and *Configure Processor...* on page 5-96.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Full descriptions of Multi-ICE configuration are given in the Multi-ICE documentation and in the online help available when the dialog is displayed.

### Remote_A configuration

To allow AXD to communicate with an Angel or EmbeddedICE target, you must configure the Remote_A connection appropriately. To configure the Remote_A connection, select the ADP target. If this is not listed, click **Add** and use the resulting browse dialog to locate and select the `remote_a.dll` file.

Select the ADP target line and click the **Configure** button to display the dialog shown in Figure 5-106.

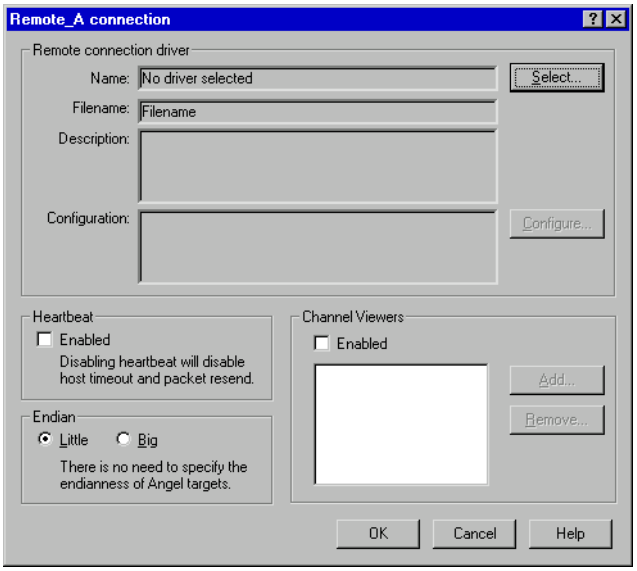



**Figure 5-106 Configuration of Remote_A connection**

The Remote_A connection dialog enables you to examine and, if necessary, change the following settings:

**Remote connection driver**

Click **Select...** to see a list of available drivers. This includes Serial, Serial /Parallel, and Ethernet drivers. Select one if you want to use it instead of the current driver. To change the settings of the currently selected driver, click **Configure...**. A dialog appears, similar to those in Figure 5-107 on page 5-95, Figure 5-108 on page 5-95, or Figure 5-109 on page 5-95.


Copyright © 1999-2001 ARM Limited. All rights reserved.

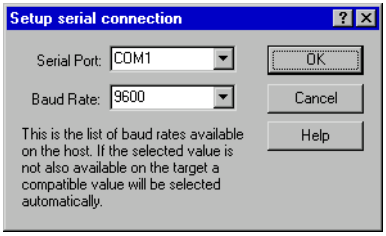



**Figure 5-107 Serial connection configuration**

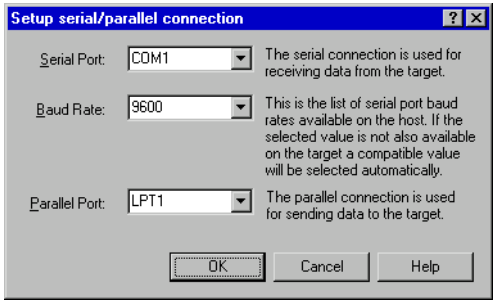



**Figure 5-108 Serial/parallel connection configuration**

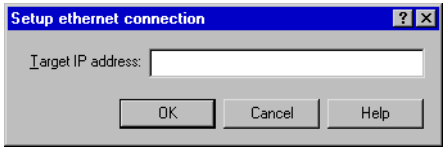



**Figure 5-109 Ethernet connection configuration**

**Heartbeat** Ensures reliable transmission by sending heartbeat messages. Any errors are more easily detected when known messages are expected regularly.

**Endian** These buttons inform the debugger that the target is operating in little-endian or big-endian mode.

- If you are using the ARMulator to simulate a processor with an MMU and you have semihosting enabled in the `.ami` configuration file, the ARMulator sets the big-endian bit in CP15. If semihosting is not enabled, the big-endian bit is not set and the processor executes in little-endian mode. In that case you must write initialization code to set the big-endian bit, or set it manually through the debugger.
- If you are using ARMulator to simulate a processor without an MMU, such as the ARM7TDMI® core, the **Endian** button sets the endianness of the target processor.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

For hardware targets such as Multi-ICE, the **Endian** button only sets the endianness expected by the debugger. You must initialize your hardware to run in the appropriate mode.

Angel automatically corrects a wrong endian target setting.

**Channel Viewers**

Channel viewers are not supported if you are running AXD under UNIX.

When you run AXD under Windows, checking **Enabled** enables you to access a displayed list of `.dll` files. Do not enable any DCC viewer from this dialog. Instead, use the AXD built-in viewer available from the **Processor Views** menu and enabled from the Processor Properties dialog. For more details see *Comms Channel processor view* on page 5-37 and *Configure Processor...*.

For information on how to configure other targets, for example ARM Agilent Debug Interface, see the documentation accompanying the product.

### 5.7.4 Configure Processor...

This menu item provides a quick way for you to display the Processor Properties dialog for the current processor. Selecting **Configure Processor...** from the **Options** menu is equivalent to right-clicking on a processor name on the **Target** tab of the Control system view and selecting **Properties** from the resulting pop-up menu. A typical Processor Properties dialog is shown in Figure 5-110.

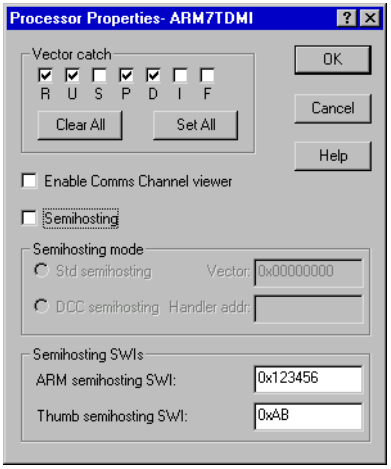



**Figure 5-110 Processor Properties dialog**


Copyright © 1999-2001 ARM Limited. All rights reserved.

The **Vector Catch** group enables you to select the exceptions that are intercepted, causing control to pass back to the debugger. The default settings of `vector_catch` are `RUsPDif`. An uppercase letter indicates an exception is intercepted. The exceptions controlled in this way are:

| | |
|---|---|
| `R` | Reset |
| `U` | Undefined Instruction |
| `S` | SWI |
| `P` | Prefetch Abort |
| `D` | Data Abort |
| `I` | *normal interrupt request* (IRQ) |
| `F` | *fast interrupt request* (FIQ) |

Each check box in the Vector Catch group indicates whether a particular exception is intercepted (checked) or ignored (blank) for the specified processor. Any changes you make become effective when you click the **OK** button. For further information see *setprocprop* on page 6-47.

The Enable Comms Channel and Semihosting selections, the Semihosting mode settings, and the Semihosting SWIs settings can interact with one another. These are governed, to some extent, by the target configuration.

Settings are disabled when it is inappropriate for you to change them. You can, however, view the current settings.

You can switch semihosting on or off using the **Semihosting** check box. When it is switched on, you can set the semihosting mode to Standard or DCC (Debug Communications Channel). If you select the DCC semihosting mode, then:

- the **Comms Channel** check box becomes disabled because the options are mutually exclusive
- you might have to change the address stored in the variable to suit the size of the target memory (see *Semihosting* on page 4-11).

The Vector field sets the value of the `$semihosting_vector` variable. See the Semihosting chapter of the *ADS Debug Target Guide* for an explanation of this variable, and for more general information on semihosting issues.

**Caution**

The Semihosting SWIs fields specify an integer number identifying the ARM and Thumb SWI numbers that are used for semihosting. You are strongly advised not to change these.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.7.5 Source Path...

Select **Source Path...** from the **Options** menu to display the Set Source Path dialog shown in Figure 5-111. This specifies the paths that are searched, and the order in which they are searched, when a source file is required.

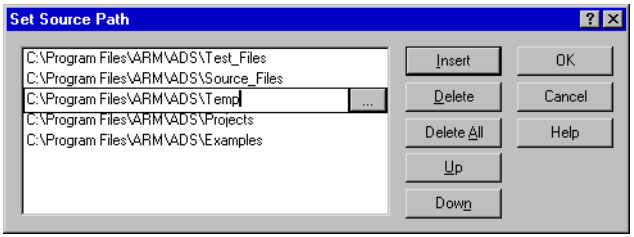



**Figure 5-111 Set Source Path dialog**

To insert a path in the list, click the **Insert** button. Either browse for the required path name or enter the full path name, then press Return. For example, you might specify `C:\Program Files\ARM\ADS\Temp` as a source path.

You can select and delete a single path name, or delete all path names. You can also select and move a path name up or down the list.

Source paths are persistent. They are saved and used in subsequent debugging sessions.

You can also set and view source paths using the command-line interface. See *sourcedir* on page 6-53, and *setsourcedir* on page 6-49.

### 5.7.6 Status Bar display control

If you click on the **Status Bar** menu item so that it is checked the status bar is displayed at the bottom of the AXD screen (see *Status bar contents* on page 5-5).

If you click the **Status Bar** menu item so that it is cleared the status bar is not displayed.

### 5.7.7 Profiling

Select **Profiling** to display a submenu, shown in Figure 5-112. This enables you to control profiling, provided you made suitable settings when you loaded the image. See *Profiling* on page 4-27 for details.

**Figure 5-112 Profiling submenu**


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 5.8 Window menu

The **Window** menu, shown in Figure 5-113, enables you to control the display of windows and icons on your screen.

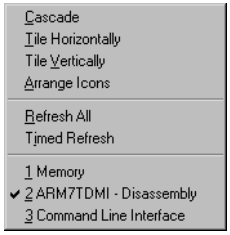



**Figure 5-113 Window menu**

Source and Disassembly views always float within the main window. All other views can be displayed in any one of three types of window:

- docked at one edge of the main window
- floating anywhere on the screen
- floating within the main window.

The **Window** menu items operate on views that are floating within the main window only. Windows that can float to any position on the screen and windows that are docked are not affected or listed.

Any cascaded or tiled windows are arranged within the screen area that remains unoccupied by any docked windows. Docked and floating windows are described in *Docked and floating windows* on page 2-10.

The **Window** menu items are described under the following headings:

- *Cascade* on page 5-100
- *Tile Horizontally* on page 5-100
- *Tile Vertically* on page 5-100
- *Arrange Icons* on page 5-100
- *Refresh All* on page 5-100
- *Timed Refresh* on page 5-100
- *List of relevant windows* on page 5-101.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 5.8.1 Cascade

**Cascade** operates on any windows set to float within the main window. They are repositioned, resized, and overlapped, to be as large as possible while still showing enough of each one to identify it and to allow you to select it. They fill most of the area of the main window that remains unoccupied by any docked windows.

### 5.8.2 Tile Horizontally

**Tile Horizontally** operates on any windows set to float within the main window. They are repositioned and resized to avoid any overlapping and to fill the area of the main window that remains unoccupied by any docked windows. The windows are made as wide as is reasonably possible within the space available, with their height restricted if necessary.

### 5.8.3 Tile Vertically

**Tile Vertically** operates on any windows set to float within the main window. They are repositioned and resized to avoid any overlapping and to fill the area of the main window that remains unoccupied by any docked windows. The windows are made as high as is reasonably possible within the space available, with their width restricted if necessary.

### 5.8.4 Arrange Icons

**Arrange Icons** arranges any windows minimized to icons along the bottom edge of the area of the main window that remains unoccupied by any docked windows.

### 5.8.5 Refresh All

The **Refresh All** menu item is useful when you are debugging a target that supports RealMonitor (see *RealMonitor support* on page 4-14). If you are debugging such a target and have several views displayed, the information shown might have been captured at various times during the debug session so can appear inconsistent.

Select **Refresh All** from the **Window** menu to update and recalculate the information in all currently displayed views.

### 5.8.6 Timed Refresh

Selecting the **Timed Refresh** menu item or clicking on the **Timed Refresh** tool is equivalent to selecting **Options → Configure Interface → Timed Refresh → Enable Timed Refresh**, and toggles on or off the automatic updating and recalculation of all displayed information at regular intervals.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

To change the refresh interval, select **Options → Configure Interface → Timed Refresh** and set Refresh Interval (tenths of seconds) to a new value.

**Timed Refresh** is useful when you are debugging a target that supports RealMonitor (see *RealMonitor support* on page 4-14). With other targets, all displayed views are refreshed each time target execution stops.

### 5.8.7 List of relevant windows

All windows that are currently floating within the main window are listed in the lower part of the **Window** menu, each window identified by the text that appears in its title bar. Refer to this list if some windows have become obscured. Select any window from the list to bring it to the front of the display.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 5.9 Help menu

The **Help** menu provides you with access to AXD online help and to details of the version of AXD that you are running. If you are licensed to use the *Trace Debug Tools* (TDT) add-on product, and your target processor supports trace, the option **TDT Help** is also available on the **Help** menu, shown in Figure 5-114.

**Figure 5-114 Help menu**

The **Help** menu items and relevant toolbar icons are described under the following headings:

- *Contents*
- *Using Help*
- *Online Books*
- *About AXD* on page 5-103
- *Toolbar icons* on page 5-103.

### 5.9.1 Contents

**Contents** displays the first page of AXD online help. You can navigate from there to any other available topic.

### 5.9.2 Using Help

**Using Help** displays instructions for various ways to obtain online help while you are using the debugger.

### 5.9.3 Online Books

**Online Books** allows you to view the ARM manuals that are published in both printed and online forms, and are complementary to online help. This is equivalent to selecting **Start → Programs → ARM Developer Suite v1.2 → Online Books**.

During ADS installation, you can choose not to install online books and PDF files. You can view online books only if they are installed. If the PDF files are not installed you can view them by reading them from the ADS installation CD-ROM.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 5.9.4 About AXD

**About AXD** displays the name, version number, and build number of the AXD software you are running.

When you have seen the details, close the dialog by clicking on either the **Close** button or the **OK** button.

### 5.9.5 Toolbar icons

Clicking on the **Query** icon is equivalent to selecting **Contents** from the **Help** menu.

Clicking on the **Query and arrow** icon changes the mouse pointer into a similar icon. Click again on any part of the display for which you want help.


Copyright © 1999-2001 ARM Limited. All rights reserved.

Copyright © 1999-2001 ARM Limited. All rights reserved.

# Chapter 6
# AXD Command-line Interface

This chapter describes the use of the *Command Line Interface* (CLI) window. It contains the following sections:


- *Command Line Window* on page 6-2
- *Parameters and prefixes* on page 6-4
- *Commands with list support* on page 6-5
- *Predefined command parameters* on page 6-6
- *Definitions* on page 6-9
- *Commands* on page 6-13.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 6.1 Command Line Window

Select **Command Line Interface** from the **System Views** menu to display the *Command Line Interface* (CLI) window. In the CLI window you can enter commands that are equivalent to many of the debugger menu items, or submit a file of such commands. This provides a reliable and consistent way for you to execute sequences of commands repeatedly.

You might use the CLI window for the following reasons:

- *As an alternative to the GUI*
- *To automate repetitive tasks*.

To display the CLI window pop-up menu, right-click in the CLI window.

You can paste text into the CLI window instead of typing it, but you must ensure that every line ends with a CR-LF pair. The final line, for example, is not executed if it does not end with CR-LF.

All commands entered, either by typing or by pasting, are added to the CLI history list.

### 6.1.1 As an alternative to the GUI

Using the GUI involves selecting items from menus. Many of these menu items correspond to commands you can enter in the CLI window.

One advantage of working in the CLI window is the ability to log all your actions in a disk file.

If any of your commands result in data being displayed by the debugger, these appear in the CLI window. You can choose whether a log file includes everything displayed in the CLI window, or your commands only.

You can use both the CLI and the GUI in a debug session. If, for example, a GUI command changes the current processor, then any CLI command that by default refers to the current processor will refer to the newly-defined processor.

### 6.1.2 To automate repetitive tasks

You can record the commands you issue in a log file (see *Command Line Interface system view pop-up menu* on page 5-65 or *record* on page 6-38). You can then easily repeat the same commands by submitting the file to the CLI using the obey command (see *obey* on page 6-36).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.1.3 CLI pop-up menu

Right-click in the CLI window to display the CLI window pop-up menu. For details refer to AXD online help or to *Command Line Interface system view pop-up menu* on page 5-65.

To display the CLI Properties dialog, shown in Figure 5-87 on page 5-67, Figure 5-88 on page 5-68, and Figure 5-89 on page 5-68, select **Properties...** from the pop-up menu.

The CLI Properties dialog enables you to set various default values so that you do not have to specify them on commands you intend to issue. It also provides an alternative method of issuing certain commands, such as toggling on or off logging or recording, or selecting files to use for those purposes.

In a few cases, this dialog provides the only method of setting values. Such values include the number of lines of disassembly or source code to display, and the number of history records visible in a view.

Click **Help** or refer to *Command Line Interface system view pop-up menu* on page 5-65 for more information about this dialog.

### 6.1.4 Command history

Your most recent commands are stored and are available for reuse. Press the up arrow and down arrow keys to move backwards and forwards through the list of recent commands. When any earlier command is displayed you can press Return to issue the command for execution.

To issue a new command similar to one you issued earlier, use the up arrow and down arrow keys to display the earlier command, then the left arrow and right arrow keys to position the cursor. Change the earlier command as required, then press Return to issue the new command.

To see the stored list of commands, press the Ctrl+Page Up key combination. If there are too many commands to display in the window, you can scroll the list. Select any displayed command and press Return to use that command as the basis for a new command.

To change the number of recent commands stored, see *Command Line Interface system view pop-up menu* on page 5-65.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 6.2 Parameters and prefixes

When entering commands, you might have to supply parameters of various types. To specify the type of a parameter, prefix its value with one of the symbols `#`, `|`, `@`, or `+`.

### 6.2.1 # parameters

After a `#` symbol the remaining character(s) must be numeric, and identify an object by its position in a list.

Before specifying an object by using a # parameter you must issue a command that displays the relevant indexed list. For commands that display indexed lists, see *Commands with list support* on page 6-5.

### 6.2.2 | parameters

Type a `|` symbol to separate a parent and a child item in a parameter that includes hierarchical levels.

You might need to include a `|` symbol when you supply a *`position`* parameter, for example, even though the symbol is not shown in the syntax description of the command.

A `|` symbol in a syntax description denotes alternatives, and you do not type it when you enter the command.

### 6.2.3 @ parameters

After an `@` symbol the remaining characters must form an expression that evaluates to an address. Usually, this kind of parameter takes one of the following forms:

- A hexadecimal value, `@0x82E0` for example.
- The name of a low-level symbol, `@Proc_4` for example.

### 6.2.4 + parameters

The `+` symbol prefixes the second parameter of a range when it is to be used as a size rather than an upper value.

### 6.2.5 Other parameters

You might need to supply other names as parameters (a file, a directory, or a debugger internal variable, for example). They do not begin with one of the symbols `#`, `|`, `@`, or `+`. For example, to display the value of the internal variable `$target_fpu`, type:

```
print $target_fpu
```

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## 6.3 Commands with list support

Several commands display lists with entries identified by an index number (starting from 1 for the first entry). You can use these index numbers to refer to specific entries.

The following indexed lists are available:

- files
- classes
- functions
- variables
- watchpoints
- breakpoints
- regbanks
- registers
- stack entries
- low-level symbols
- processors
- images.

Commands that display these indexed lists and commands that accept indexed entries from these lists are described in *Commands* on page 6-13.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 6.4 Predefined command parameters

Several commands take parameters in the form of text strings, but a very few predefined values are the only ones you are allowed to supply. For example, where *toggle* is specified as a parameter, you can enter either the string `on` or the string `off`. Any other value for this parameter is invalid.

In the alphabetical list of *Commands* on page 6-13, the parameters printed in *italics* are those that you replace with the value you require when you issue the command.

These parameters are not case-sensitive. You can freely mix uppercase and lowercase characters. The parameters for which you must specify certain values only are described in the following sections:

- *format*
- *asm*
- *instr* on page 6-7
- *step* on page 6-7
- *memory* on page 6-7
- *scope* on page 6-7
- *toggle* on page 6-8.

### 6.4.1 format

The *format* parameter must be set to the name or index number of an existing format. To display a list of all currently available formats, refer to *listformat* on page 6-32.

Use this parameter to specify how values are displayed. For example, each line in a memory listing shows the contents of 16 bytes of memory, grouped into 4, 8, or 16 values (see *memory* on page 6-7). The setting of the *format* parameter in the `memory` command determines whether each value is displayed in hexadecimal, decimal, octal, binary, or any other available format.

You can also use the *format* parameter to specify the default display format for registers, memory, or watchpoints. The default setting of *format* is shown on the **Format** tab of the CLI Properties dialog or by the `format` command.

For more information about formats, see *Data formatting* on page 4-16.

### 6.4.2 asm

The *asm* parameter must be set to `ARM`, `Thumb`, `ByteCode` or `auto`.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

ARM instructions occupy 32 bits and Thumb instructions occupy 16 bits. ARM C and C++ compilers can generate either ARM or Thumb code. Use the *asm* parameter to specify that the code being debugged contains ARM code or Thumb code, or Jazelle code, or that the debugger must make the setting itself (`auto`). You usually have to specify the instruction type only when the code was built without debug information.

The setting of *asm* is shown in the Instruction Size field of the CLI Properties dialog.

### 6.4.3 instr

The *instr* parameter must be set to `line` or `instr`.

This parameter determines whether a step consists of a line of source code (`line`) or an assembler instruction (`instr`).

You can examine or change the setting of *instr* in the Step size field of the CLI Properties dialog **General** tab, or with the `stepsize` CLI command.

### 6.4.4 step

The *step* parameter, if specified, must be set to `in` or `out`.

This affects the way an instruction calling a function is processed. If you specify `in`, the step proceeds only to the first executable instruction in the called function. If you specify `out`, the step includes execution of the called function and proceeds to the instruction at which execution returns to the calling program. If you omit the *step* parameter, execution steps over a line or instruction.

### 6.4.5 memory

The *memory* parameter must be set to `8`, `16`, or `32`.

| | |
|---|---|
| `8` | Displays memory in 8-bit bytes. |
| `16` | Displays memory in 16-bit halfwords. |
| `32` | Displays memory in 32-bit words. |

The setting of *memory* is shown in the Size field of the CLI Properties dialog.

To specify the format for displaying values, see *format* on page 6-6.

### 6.4.6 scope

The *scope* parameter must be set to `class`, `global`, or `local`.

This parameter specifies that any context variables displayed by the associated command are those scoped to class, global, or local, respectively.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.4.7 toggle

The *`toggle`* parameter must be set to `on` or `off`.

This parameter switches the associated command on or off.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 6.5 Definitions

With most commands you have to specify parameters that define, for example, a processor, file, position, address, or format. This section lists these definitions and explains how to use them as command parameters:

*asm*
Denotes that assembler instructions are ARM (32-bit), Thumb (16-bit) or Jazelle (8-bit). You must specify `ARM`, `Thumb`, `ByteCode` or `auto`. If you specify `auto`, the debugger determines the correct setting itself when possible.

*breakpoint*
You specify a breakpoint as its index in the breakpoint list, in the form of a value prefixed by `#`.

*class*
You can identify a class by:
- the class name which can include the name of an image, separated from the class name by a vertical bar, in the form *image*|*class*
- the index of the class in the current class list, in the form of a value prefixed with #.

*context*
You can specify a context by specifying a stack entry, in the form of a value prefixed by `#`.

*expr*
An expression is either a numerical value or an expression that evaluates to a numerical value.

*file*
You can identify a file by:
- its filename
- the index of the file in the current file list, in the form of a value prefixed with `#`
- the globally unique identifier of the file as shown in the output of a files command
- null, the current file is used.

*format*
Denotes the format in which the contents of memory, registers, or variables are displayed. You must specify the name or index number of an available format.

*image*
You can identify an image by:
- the name of the image
- the index of the image in the current image list, in the form of a value prefixed with `#`
- the globally unique identifier of the image as shown in the output of an images command


Copyright © 1999-2001 ARM Limited. All rights reserved.

- null (defaults to the image associated with the current processor).

*index*
You can refer to items in a list by specifying their position in the list. For example, this gives you a convenient way of referring to a watchpoint in commands such as `clearwatch`.

*instr*
You must specify either `instr` to define a step as one instruction or `line` to define a step as one line of source code.

*ipvariable*
Denotes any one of a group of variables that define image-related properties. The variables currently supported are:

`cmdline`
This variable holds the parameter passed to the image when execution starts. If the image requires multiple parameters, enclose the whole string in quotes ("`...`").

*memory*
Denotes that memory is to be displayed in bytes, halfwords, or words. You must specify `8`, `16`, or `32`.

*position*
To specify a position in a source file, use vertical bar separators as in *`image|file|line`*. If you omit the image name, the image associated with the current processor is assumed.

A position might also be a location within an executable image. In this case you can specify it in the form *`image|@address`*.

A position can also be inferred from many debug objects, such as breakpoints or low-level symbols. You can therefore specify a position as an index of a position-based object in the last displayed list of these objects. Specify the index as a value prefixed by `#`.

*ppvariable*
Denotes any one of a group of variables that define processor-related properties. The variables currently supported are:

`vector_catch`
Defines which exceptions in the processor are intercepted by the debugger. For details see *Processor pop-up menu* on page 5-50 and *setprocprop* on page 6-47.

`comms_channel`
Enables or disables the communications channel.

`semihosting_enabled`
Enables or disables semihosting, as follows:

`0` semihosting disabled

`1` standard semihosting enabled

`2` DCC semihosting enabled (applies only to Multi-ICE or ARM Agilent Debug Interface, the add-on product to ADS).


Copyright © 1999-2001 ARM Limited. All rights reserved.

`semihosting_vector`
: Defines handler address. Applies to Multi-ICE only.

`semihosting_dcchandler_address`
: Defines handler address. Applies to Multi-ICE only.

`arm_semihosting_swi`
: Defines ARM software interrupt number reserved for semihosting.

`thumb_semihosting_swi`
: Defines Thumb software interrupt number reserved for semihosting.

*processor*
: You can identify a processor by:

- the name of the processor
- the index of the processor in the current processor list, in the form of a value prefixed with `#`
- the globally unique identifier of the processor as shown in the output of a `processors` command
- null (defaults to the current processor).

*regbank*
: You can identify a register bank by:

- The name of the register bank. This is processor-dependent. Use the `regbanks` command to generate a register bank list. Examples of register banks are:
  - `Current`
  - `User` or `System` or `User/System`
  - `IRQ`
  - `FIQ`
  - `SVC`
  - `Abort`
  - `Undef`
  - `EICE`
  - `EICE Watch 0`
  - `EICE Watch 1`

  For example, to write to the Address register of Watchpoint Unit 0 use:

  ```
  sreg "EICE Watch 0|Address Value" 0x00008098
  ```
- The index of the register bank in the register bank list, in the form of a value prefixed with `#`.
- The globally unique identifier of the register bank shown in the register bank list.


Copyright © 1999-2001 ARM Limited. All rights reserved.

*register* You can use the `registers` command to list the registers in a register bank. You can identify a register by:

- the name of the register
- the index of the register in a register list generated by the `registers` command, in the form of a value prefixed with `#`.

*scope* Denotes which context variables to display, based on their scope. You must specify `class`, `local`, or `global`.

*step* This controls the amount of processing that takes place following an instruction that calls a function. You must specify this as `in` or `out`, or omit it. If omitted, the step is interpreted as step over line or instruction.

*string* You specify a text string enclosed in quotes (“`...`”).

*toggle* Where this parameter is allowed, you can use it to switch on or off certain properties. You must specify either `on` or `off`.

*value* You specify a numeric value.

*watchpoint* You specify a watchpoint as its index in the watchpoint list, in the form of a value prefixed by `#`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 6.6 Commands

This section lists in alphabetical order all the commands that you can issue using the command-line interface. Refer to *Definitions* on page 6-9 for descriptions of parameters used with many of these commands.

In the syntax definition of each command, square brackets (`[...]`) enclose optional parameters and a vertical bar (`|`) separates alternatives from which you choose one. Do not type the square brackets or the vertical bar.

You might need to type vertical bars when entering hierarchical values, for example *`imagename`*`|@`*`address`*. for a *`position`* parameter.

Replace parameters printed in *`italics`* with the value you require.

When you supply more than one parameter, use a comma or a space as a separator. The syntax definitions and examples in this chapter use a space.

If a parameter is a name that includes spaces, enclose it in quotation marks.

If you want to enter a command that is similar to one you have previously entered, use the up and down arrow keys to retrieve the earlier command, then use the left and right arrow keys to position the cursor where you want to change the command. The number of commands is defined in the history list.

Ctrl+Page Up shows a complete history list of commands.

Where lines of output are described, `<tab>` indicates that the items are displayed in columns.

A few command descriptions include an alias for the command. You can use either the command or its alias. Aliases are supported because you might be familiar with their use in armsd, or use these forms of the commands in existing script files.

### 6.6.1 addsourcedir

No longer a valid command. See *setsourcedir* on page 6-49 and *sourcedir* on page 6-53.

### 6.6.2 backtrace

See *stackentries* on page 6-53.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.3 break

If you supply no parameters, the `break` command lists all the breakpoints that are currently set. Each breakpoint is shown on a separate line, after a first line containing the headings for the following columns:

`Index` The position in the list. This gives you a convenient way of referring to a breakpoint in commands such as `clearbreak`.

`State` Displays an X if the breakpoint is currently disabled.

`Position` This shows the fully-qualified source code filename in brackets, a colon, and the source code line number at which the breakpoint is set. It also shows, in square brackets, the corresponding memory address.

`Count` Two numbers are shown as `X/Y`. `X` is the number of times execution has arrived at the breakpoint since the last time the breakpoint was triggered. `Y` is the number of times execution has to arrive at the breakpoint to trigger it.

`Size` This shows whether the breakpoint is ARM-sized (32 bits) or Thumb-sized (16 bits). The breakpoint size can usually be detected automatically, in which case `AUTO` is shown.

`Condition` Any condition that you have specified that must also be satisfied before the breakpoint can be triggered is shown here. This must be a boolean expression.

`Additional` The final column displays additional information. This can include one or more of the following:

- `Processor` Identifies the processor in which the breakpoint is set.
- `S/HW` Shows whether the breakpoint is implemented in hardware or software.
- `(ID)` A hardware breakpoint can have a hardware resource identifier. These identifiers are shown here.
- `Action` This shows whether the action taken when the breakpoint is triggered is to stop execution of the target (`Break`) or to log the event (`Log`).

If you supply parameters, the command creates and sets a new breakpoint so that execution continues until the specified address is visited for the *n*th time. If you do not specify a value for *n*, a default value of 1 is assumed, so execution stops every time the address is visited.

The shorthand form of the `break` command is `br`.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Syntax

`br[ expr|posn [ n]]`

where:

*expr|posn* Is either an expression or a position that defines where a new breakpoint is to be created.

*n* Specifies the number of times execution must arrive at the breakpoint in order to trigger it. The default value is 1.

### Examples

`br 0x8000` Sets a breakpoint at address `0x8000`

`br @main` Sets a breakpoint on `main`.

`br c:\test\main.c|130 100`

Sets a breakpoint on line 130 of file `main.c`, requiring 100 arrivals to trigger it.

`br #5|150` Sets a breakpoint at line 150 of file number 5. The index #5 must have been obtained using the `files` command.

When you have created a new breakpoint, you can change its properties with the `SetBreakProps` command. See *setbreakprops* on page 6-44.

A sample listing is shown in Example 6-1.

**Example 6-1 Break listing**

```
Debug >br
Index     Position                      Count  Size  Condition         Additional
#1        [0x000084EC]{dhry_1.c:149}    0/1    AUTO  N/A               ARM7T_1 HW(-1) Log: Point A hit
        Position: [0x000084EC]{C:\Program Files\ARM\ARM Developer Suite\Examples\dhry\dhry_1.c:149}
#2        [0x000084F0]{dhry_1.c:150}    0/750  AUTO  N/A               ARM7T_1 HW(-1) Break
        Position: [0x000084F0]{C:\Program Files\ARM\ARM Developer Suite\Examples\dhry\dhry_1.c:150}
#3        [0x00008290]{dhry_1.c:91}     0/1    AUTO  N/A               ARM7T_1 HW(-1) Break
        Position: [0x00008290]{C:\Program Files\ARM\ARM Developer Suite\Examples\dhry\dhry_1.c:91}
Debug >
```

**Note**

To set complex breakpoints, use the `setbreakprops` command (see *setbreakprops* on page 6-44) or select **Breakpoints...** from the **System Views** menu.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.4 cclasses

The `cclasses` command lists all the classes in the specified class in the currently loaded image. Each class is shown on a separate line, in the following format:

```
index<tab>classname
```

The position in this list, `index`, gives you a convenient way of referring to a class of classes.

The shorthand form of the `cclasses` command is `ccl`.

#### Syntax

`ccl` *`class`*

#### Example

`ccl testclass` Displays subclasses of `testclass`.

### 6.6.5 cfunctions

The `cfunctions` command lists all the functions in the specified class. Each variable is shown on a separate line, in the following format:

```
index<tab>functionname (parameterlist)
```

The position in this list, `index`, gives you a convenient way of referring to a class function.

The shorthand form of the `cfunctions` command is `cfu`.

#### Syntax

`cfu` *`class`*

#### Example

`cfu #2` Displays functions in the class identified by index number 2. The index must have been obtained using the `classes` command.

### 6.6.6 classes

The `classes` command lists all the classes in the specified image, or in the current image if you do not specify an image. Each class is shown on a separate line, in the following format:

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

`index<tab>classname`

The position in this list, `index`, gives you a convenient way of referring to a class.

The shorthand form of the `classes` command is `cl`.

#### Syntax

`cl[ image]`

### 6.6.7 clear

The `clear` command clears the command-line window.

The shorthand form of the `clear` command is `clr`.

#### Syntax

`clr`

### 6.6.8 clearbreak

The `clearbreak` command unsets and deletes a specified breakpoint or all current breakpoints. See *break* on page 6-14 for a description of how to refer to a breakpoint.

The shorthand form of the `clearbreak` command is `cbr`.

#### Alias

`unbreak` is an alias for `clearbreak`.

#### Syntax

`cbr breakpoint|all`

#### Examples

| | |
|---|---|
| `cbr #2` | Clears breakpoint number 2. The index #2 must have been obtained using the `break` command. |
| `unbreak all` | Clears all current breakpoints. The parameter `all` is case-sensitive. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.9 clearstat

The `clearstat` command deletes the set of accumulated statistics at the specified reference point.

The shorthand form of the `clearstat` command is `cstat`.

#### Syntax

`cstat` *`referencepoint`*

where:

*`referencepoint`*

Specifies the set of statistics you want to delete. You must specify a reference point name. This name is case-sensitive. If the name contains spaces, enclose it in double quotes.

#### Examples

`cstat rp001` Deletes the set of statistics at reference point `rp001`.

`cstat "Ref Point 2"`

Deletes the set of statistics at reference point `Ref Point 2`.

You cannot delete the line of statistics that has the reference point name `$statistics`.

If you specify a reference point that does not exist, an error message is displayed.

See also *statistics* on page 6-54 for a description of how to add a new reference point or display all reference points.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.10 clearwatch

The `clearwatch` command unsets and deletes a specified watchpoint or all current watchpoints. See *watchpt* on page 6-59 for a description of how to refer to a watchpoint.

The shorthand form of the `clearwatch` command is `cwpt`.

#### Alias

`unwatch` is an alias for `clearwatch`.

#### Syntax

`cwpt` *`watchpoint`*`|all`

#### Examples

| | |
|---|---|
| `cwpt #2` | Clears watchpoint number 2. The index #2 must have been obtained using the `watchpt` command. |
| `unwatch all` | Clears all current watchpoints. The parameter `all` is case-sensitive. |

### 6.6.11 comment

The `comment` command sends the specified character string to the current log file. If logging is not taking place this command has no effect.

The shorthand form of the `comment` command is `com`.

#### Syntax

`com` *`string`*


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.12 context

If you do not supply a parameter, the `context` command displays details of the current context, as follows:

```
Image: imagename|@address File: sourcefilename|linenumber
```

If you specify a stack entry, the `context` command sets the current context to that of the stack entry you specify. See *stackentries* on page 6-53 for further information on stack entries.

This command does not change the execution context. It enables you to browse through all the available contexts of the current debug session and examine context-related variables.

The shorthand form of the `context` command is `con`.

#### Syntax

`con[ context]`

#### Example

| | |
|---|---|
| `con #2` | Sets the current context to that of stack entry number 2. The index #2 must be obtained using the `stackentries` command. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.13 convariables

The `convariables` command displays the name, type, and value of all variables valid in the current or specified context and in the specified scope. If you do not specify a scope, then class, global, and local variables are listed.

The shorthand form of the `convariables` command is `convar`.

#### Syntax

`convar[ context][ scope][ format]`

where:

*`context`* Specifies the context of the variables you want to list, the default being the current context (see *stackentries* on page 6-53).

*`scope`* Can be set to `class`, `global`, or `local` (see *scope* on page 6-7).

*`format`* Specifies the format in which the contents of the variables are listed, if this is different from the default format (see *format* on page 6-6).

#### Examples

`convar #1 dec`

Displays the global, class, and local variables in the context of stack entry number 1, in decimal format. Index #1 must be obtained with the `stackentries` command.

`convar local` Displays the local variables in the current context, in hexadecimal format.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.14 cvariables

The `cvariables` command lists all the variables in the specified class in the currently loaded image. Each variable is shown on a separate line, in the following format:

```
index<tab>variablename<tab>type
```

The position in this list, `index`, gives you a convenient way of referring to a class variable.

The shorthand form of the `cvariables` command is `cva`.

#### Syntax

`cva` *`class`*

#### Examples

`cva testclass`

Displays the class variables of `testclass`.

`cva #1`

Displays the class variables of the class identified by index number 1. The index must be obtained using the `classes` command.

### 6.6.15 dbginternals

The `dbginternals` command displays the debugger internal variables of the current target. These are the same variables as those displayed when you select **Debugger Internals** from the **System Views** menu. Each variable is shown on a separate line, in the following format:

```
variablename<tab>value
```

The shorthand form of the `dbginternals` command is `di`.

#### Syntax

`di`


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.16 disassemble

The `disassemble` command disassembles and displays lines of assembler code that correspond to the contents of the specified area of memory.

The shorthand form of the `disassemble` command is `dis`.

#### Alias

`list` is an alias for `disassemble`.

#### Syntax

`dis` *`expr1`*`[ [+]`*`expr2`*`[ `*`asm`*`]]`

where:

*`expr1`* Is an expression that evaluates to the starting address of the area of memory you want to see disassembled.

*`expr2`* Is an expression that either evaluates to the end address of the area of memory you want to see disassembled or, if preceded by `+`, evaluates to the number of bytes you want disassembled. If a value is not supplied on the command line, the value from the Bytes to display property box is used.

*`asm`* Can be set to `ARM`, `Thumb`, `ByteCode` or `auto` (see *asm* on page 6-6). If not specified, the current value of the Instruction Size field of the CLI properties dialog is used.

#### Example

```
dis 0x8200 +64 ARM
```

Displays disassembled instructions that represent the ARM code currently stored in the 64 bytes of memory starting at address `0x8200`.

```
dis 0x8000 +10 ByteCode
```

Displays disassembled instructions that represent the Jazelle code currently stored in the 10 bytes of memory starting at address `0x8000`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.17 echo

The `echo` command enables you to choose whether CLI commands read from an Obey file are displayed in the CLI system view. Because you can log whatever is displayed in the CLI system view, this command also determines whether CLI commands read from an Obey file are logged.

If an Obey file includes an `echo` command, the new setting is effective only while commands from that Obey file are being executed. The `echo` setting then reverts to the state it was in when the Obey process began.

Using the `echo` command is equivalent to checking or unchecking the **Echo** check box in the CLI Properties dialog.

There is no shorthand form of the `echo` command.

#### Syntax

```
echo on|off
```

where:

| | |
|---|---|
| `on` | Means that CLI commands subsequently read from an Obey file appear in the CLI system view. This is the default setting. |
| `off` | Means that CLI commands subsequently read from an Obey file do not appear in the CLI system view. |

### 6.6.18 examine

See *memory* on page 6-35.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.19 files

The `files` command lists all the source files that have contributed debug information to the specified image, or to the current image if you do not specify an image. Each source file is shown on a separate line, in the following format:

```
index<tab>ID<tab>filename
```

This means that you can refer to a source file in any one of three ways:

| | |
|---|---|
| `index` | The position in this list. |
| `ID` | The identifier of the source file. |
| `filename` | The name of the source file. |

The shorthand form of the `files` command is `fi`.

#### Syntax

```
fi[ image]
```

### 6.6.20 fillmem

The `fillmem` command fills the specified area of memory with the specified value repeated sufficient times. If the size of the area to be filled is not an exact multiple of the size of the value being written, some bytes remain unchanged at the end of the area. The value written (repeatedly) to memory is the value you specify, padded with leading zeros or truncated if necessary to achieve the size you specify with the *`memory`* parameter.

The shorthand form of the `fillmem` command is `fmem`.

#### Syntax

```
fmem expr1 [+]expr2 value[ memory]
```

where:

| | |
|---|---|
| *`expr1`* | Specifies the starting address of the area of memory to be filled. |
| *`expr2`* | Specifies either the end address or, if preceded by `+`, the number of bytes of memory to be filled. |
| *`value`* | Specifies what is to be written to memory. |
| *`memory`* | Can be set to `8`, `16`, or `32`, and determines whether *`value`* should be evaluated to an 8-bit, a 16-bit, or a 32-bit value (see *memory* on page 6-7). |


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Example

```
fmem 0x83A4 +20 0x61626364 32
```

Overwrites the 20 bytes of memory starting at address `0x83A4` with the 4-byte value `0x61626364` repeated five times. To see the effect of this and other commands, load an image, open a Memory processor view big enough to see about 16 lines and set its starting address to `0x8300`. Then in the CLI system view perform the examples given for the commands fillmem, savebinary, reload, and loadbinary.

## 6.6.21 findstring

The `findstring` command searches for the specified string in the specified area of memory or, by default, in the whole available memory range. The command displays messages giving the starting address of every occurrence found of the specified value.

If you view the contents of memory with size set to more than 8 bits, it is possible for bytes to be displayed in an order different from that in which they are stored (as a result of the endian setting). The `findstring` command always tests consecutive memory locations, regardless of how the contents of those locations might be displayed.

The shorthand form of the `findstring` command is `fds`.

### Syntax

`fds` *`string`*`[[` *`low-expr`*`][ [+]`*`high-expr`*`]]`

where:

*`string`* Specifies the string you are seeking.

*`low-expr`* Is an expression that evaluates to the memory address where the search is to begin.

*`high-expr`* Is an expression that evaluates to the memory address where the search is to end or, if preceded by `+`, the number of bytes of memory to search.

### Example

```
fds "cb" 0x8300 0x8400
```

Reports finding the specified string at five addresses within the specified range if you have performed the fillmem example. The order in which the bytes you entered in the fillmem example are stored depends on the endian setting of the target. This `fds` example assumes they were stored in the order `0x64 0x63 0x62 0x61` (`"dcba"`).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.22 findvalue

The `findvalue` command searches for the specified value in the specified area of memory or, by default, in the whole available memory range. The command displays messages giving the starting address of every occurrence found of the specified value.

If you view the contents of memory with size set to more than 8 bits, it is possible for bytes to be displayed in an order different from that in which they are stored (as a result of the endian setting). The `findvalue` command always tests consecutive memory locations, regardless of how the contents of those locations might be displayed.

The shorthand form of the `findvalue` command is `fdv`.

#### Syntax

`fdv` *`valexpr`*`[[` *`low-expr`*`][ [+]`*`high-expr`*`]]`

where:

*`valexpr`* Is an expression that evaluates to the value you are seeking.

*`low-expr`* Is an expression that evaluates to the memory address where the search is to begin.

*`high-expr`* Is an expression that evaluates to the memory address where the search is to end or, if preceded by `+`, the number of bytes of memory to search.

#### Example

```
fdv 0x6362 0x8300 0x8400
```

Reports finding the specified value at five addresses within the specified range if you have performed the fillmem example. The order in which the bytes you entered in the fillmem example are stored depends on the endian setting of the target. This fdv example assumes they were stored in the order `0x64 0x63 0x62 0x61` (`"dcba"`).


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.23 format

The `format` command sets the default format to be used for displaying data in the CLI system view or, if issued with no parameters, reports the current display format.

The shorthand form of the `format` command is `fmt`. See also, *importformat* on page 6-31 and *listformat* on page 6-32.

#### Syntax

`fmt [`*`format_name`*`[ `*`control_string`*`]]`

where:

*`format_name`*
Defines the format to be used, in any of the following forms:

- *`#n`* Where *n* is the index number of the format as shown in the last displayed format list (see *listformat* on page 6-32).
- `RDIName` As shown in the last displayed format list.
- `ShortName` As shown in the last displayed format list, or on the **Format** tab of the CLI Properties dialog.

*`control_string`*
Defines any associated control string required by the specified format. For example, with a Q-format as the first parameter, a printf control string as the second parameter defines how values are displayed. (Q-format is currently the only supplied format that can take a further control string.)

#### Examples

`fmt #3` Sets format number 3 as the default for displays in the CLI system view.

`fmt Q3.29 %12.6f`
Interprets data in Q3.29 format and displays the values in 12.6f format.

`fmt 0x%4x` Uses printf with `0x%4x` as its control string.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.24 functions

The `functions` command lists all the functions in the specified image, or of the current image if you do not specify an image. Each function is shown on a separate line, in the following format:

```
index<tab>functiontype functionname (ParameterList)
```

The position in this list, `Index`, gives you a convenient way of referring to a function.

The shorthand form of the `functions` command is `fu`.

#### Syntax

```
fu[ image]
```

### 6.6.25 getfile

See *loadbinary* on page 6-33.

### 6.6.26 go

See *run* on page 6-40.

### 6.6.27 help

The `help` command invokes AXD online help.

The shorthand form of the `help` command is `hlp`.

#### Syntax

```
hlp
```


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.28 images

The `images` command lists all the images currently loaded on the target. Each image is shown on a separate line, in the following format:

```
index<tab>ID<tab>imagename
```

This means that you can refer to an image in any one of three ways:

| | |
|---|---|
| `index` | The position in this list. |
| `ID` | The identifier of the image. |
| `imagename` | The name of the image. |

For an example of a command that can refer to an image see *reload* on page 6-40.

The shorthand form of the `images` command is `im`.

#### Syntax

```
im
```

### 6.6.29 imgproperties

The `imgproperties` command displays internal variables related to the specified image, or to the currently loaded image if you do not specify an image. See *Definitions* on page 6-9 for a list of image-related internal variables that you can set, and *setimgprop* on page 6-45 for details of a command you can use to set them.

Each variable is shown on a separate line, in the following format:

```
ipvariable:<tab>value
```

The shorthand form of the `imgproperties` command is `ip`.

#### Syntax

```
ip[ image]
```

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.30 importformat

The `importformat` command searches a specified file for any valid format descriptions. Any valid formats found, that do not conflict with internal formats listed under RDINames (see *listformat* on page 6-32), are added to the list of available formats. A parameter enables you to specify what happens in the event of a conflict of format names.

Files most likely to contain format descriptions are supplementary display modules, having a `.sdm` filename extension, see Appendix C *Supplementary Display Module Formats*.

The shorthand form of the `importformat` command is `impfmt`.

See also *format* on page 6-28.

#### Syntax

`impfmt` *`sdm_file`*`[` *`fail_action`*`]`

where:

*`sdm_file`* Specifies a supplementary display module (a `.sdm` file) that contains format descriptions.

*`fail_action`* Specifies the action to take if an imported format description conflicts with an existing debugger internal format. You can specify any of the following actions:

- `fail` This is the default action, and returns an error message reporting the conflict.
- `msgbox` A message box prompts you to select the fail, ignore, or replace option.
- `replace` The new format definition replaces the existing one of the same name.
- `ignore` The new definition is ignored, and the existing definition remains unchanged.

### 6.6.31 let

See *setwatch* on page 6-50.

### 6.6.32 list

See *disassemble* on page 6-23.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.33 listformat

The `listformat` command displays a list of available formats, each on a separate line in the following format:

```
Index<tab>ShortName<tab>RDIName
```

The position in this list, `Index`, gives you a convenient way of referring to a format.

The format name shown under ShortName is the name that appears in the various format submenus. Since you can name and define more formats this name cannot be guaranteed to be unique. The format name shown under RDIName is the system-wide unique name of each available format.

The shorthand form of the `listformat` command is `lsfmt`.

See also *format* on page 6-28.

#### Syntax

`lsfmt[ n]`

where:

*n* Is an optional number specifying a number of bits. If you specify 16, for example, then the command lists only those formats that are appropriate for displaying 16-bit values. If you do not supply a data item size, then the command lists all available formats.

### 6.6.34 load

The `load` command loads the contents of the specified image file onto the specified processor. If you do not specify a processor, the command loads the image onto the current processor.

The shorthand form of the `load` command is `ld`.

#### Syntax

`ld file[ processor]`

where:

*file* Specifies the file containing the image you want to load.

*processor* Specifies the processor onto which you want to load the image.

An image loaded by the `load` command has a default breakpoint set at the first executable instruction in `main()`.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.35 loadbinary

The `loadbinary` command reads the specified file and loads its contents into target memory, starting at the specified address.

The shorthand form of the `loadbinary` command is `lb`.

#### Alias

`getfile` is an alias for `loadbinary`.

#### Syntax

`lb` *`file addrexpr`*

where:

*`file`* Specifies the file containing the data to be loaded.

*`addrexpr`* Is an expression that evaluates to a memory address.

#### Example

```
lb sbtest.bin 0x8300
```

Copies the contents of a file called `sbtest.bin` into an area of memory starting at address `0x8300`. To see the effect of this command, load an image, open a Memory processor view and set its starting address to `0x8300`. Then in the CLI system view perform the examples given for the commands fillmem, savebinary, reload, and loadbinary.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.36 loadsession

The `loadsession` command loads any earlier debug session that was saved in a session file that is still available. (To save a debug session, see *savesession* on page 6-43.)

The shorthand form of the `loadsession` command is `lss`.

#### Syntax

`lss` *`file`*

where:

*`file`* Specifies the session file to load.

### 6.6.37 loadsymbols

The `loadsymbols` command loads debug information from the specified file onto the specified processor, or onto the current processor if you do not specify a processor.

The shorthand form of the `loadsymbols` command is `lds`.

#### Alias

`readsyms` is an alias for `loadsymbols`.

#### Syntax

`lds` *`file`*`[` *`processor`*`]`

where:

*`file`* Specifies the file containing the symbols you want to load.

*`processor`* Specifies the processor onto which you want to load the symbols.

### 6.6.38 log

The `log` command starts or stops logging the contents of the CLI window to a disk file. If you supply no parameter, logging stops. If you supply a filename, logging starts in the specified file and any existing log file is closed. See also *record* on page 6-38.

There is no shorthand form of the `log` command.

#### Syntax

`log[` *`file`*`]`

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.39 lowlevel

The `lowlevel` command lists all the low-level symbols associated with the specified image, or with the current image if you do not supply a parameter. Each low-level symbol is shown on a separate line, in the following format:

```
index<tab>address<tab>symbolname
```

The position in this list, `index`, gives you a convenient way of referring to a low-level symbol in other commands.

The shorthand form of the `lowlevel` command is `lsym`.

#### Syntax

```
lsym[ image]
```

### 6.6.40 memory

The `memory` command displays the specified area of memory according to the specified size and format parameters, or using default size and format settings if you do not supply them (to set default values, use either the `format` command or the CLI Properties dialog). Each line displayed shows the contents of 16 bytes of memory, as follows:

```
address<tab>formattedvalues<tab>ASCIIequivalents
```

The shorthand form of the `memory` command is `mem`. The ASCII equivalent is based on the 8-bit value.

#### Alias

`examine` is an alias for `memory`.

#### Syntax

```
mem expr1[ [+]expr2[ memory[ format]]]
```

where:

*expr1* Is an expression that evaluates to the starting address of the area of memory that you want to examine.

*expr2* Is an expression that either evaluates to the end address of the area of memory that you want to examine or, if preceded by a `+`, evaluates to the number of bytes that you want to examine. If expr2 is not present, the number of bytes displayed uses the value in the Bytes to display dialog box.

*memory* Can be set to `8`, `16`, or `32` (see *memory* on page 6-7).


Copyright © 1999-2001 ARM Limited. All rights reserved.

*format* Can be set to the RDI name as shown in the last displayed format list or to the index number of any available format (see *format* on page 6-6).

#### Example

```
mem 0x8300 +256 8 hex
```

Displays 16 lines, each showing the address of the first byte, the contents of 16 bytes, and their ASCII equivalents.

### 6.6.41 obey

The `obey` command executes the list of CLI commands contained in the specified file.

There is no shorthand form of the `obey` command.

#### Syntax

`obey` *`file`*

where:

*`file`* Identifies a file containing valid CLI commands, each separated by a carriage return, with the end of file at the beginning of a new line.

### 6.6.42 parse

The `parse` command sets the parsing state on or off according to the supplied parameter. You must normally leave `parse` set to its default value of `on` so that commands are checked for valid syntax before being translated into internal commands.

The shorthand form of the `parse` command is `par`.

#### Syntax

`par` *`toggle`*

where:

*`toggle`* Must be set to `on` or `off`.

### 6.6.43 print

See *watch* on page 6-59.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.44 processors

The `processors` command lists all the processors available on the current target. Each processor is shown on a separate line, in the following format:

```
index<tab>ID<tab>procname
```

This means you can refer to a processor in any one of three ways:

| | |
|---|---|
| `index` | The position in this list. |
| `ID` | The identifier of the processor. |
| `procname` | The name of the processor. |

For examples of commands in which you might need to refer to a processor see *stop* on page 6-56 and *run* on page 6-40.

The shorthand form of the `processors` command is `proc`.

#### Syntax

```
proc
```

### 6.6.45 procproperties

The `procproperties` command displays internal variables related to the debug target of the specified processor, or to the current processor if you do not specify a processor. The command displays variables such as the vector catch settings, semihosting status, and the status of the debug communications channel. Each variable is shown on a separate line, in the following format:

```
ppvariable<tab>value
```

The shorthand form of the `procproperties` command is `pp`.

#### Syntax

```
pp[ image]
```

### 6.6.46 putfile

See *savebinary* on page 6-42.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.47 quitdebugger

The `quitdebugger` command ends execution of AXD.

The shorthand form of the `quitdebugger` command is `quitd`.

#### Syntax

```
quitd
```

### 6.6.48 readsyms

See *loadsymbols* on page 6-34.

### 6.6.49 record

The `record` command starts or stops the logging of commands (only) to a disk file. If you supply no parameter, logging stops. If you supply a filename, logging starts in the specified file and any existing log file is closed. See also *log* on page 6-34.

The shorthand form of the `record` command is `rec`.

#### Syntax

```
rec[ file]
```

### 6.6.50 regbanks

The `regbanks` command lists all the register banks associated with the specified processor, or with the current processor if you do not supply a parameter. Each register bank is shown on a separate line, in the following format:

```
index<tab>ID<tab>regbankname
```

The position in this list, `index`, gives you a convenient way of referring to a register bank in other commands. For this command, `ID` is given without a leading `#` character.

The shorthand form of the `regbanks` command is `regbk`.

#### Syntax

```
regbk[ processor]
```

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.51 registers

The `registers` command lists all the registers and their values in the specified register bank. The register bank name is displayed on the first output line, and column headings on the second. Each register is then shown on a separate line, in the following format:

```
index<tab>regname<tab>regvalue
```

The index value given in this list enables you to specify individual registers in other commands. See *setreg* on page 6-48, for example.

The value of each register is shown in its default format unless you specify a format.

The shorthand form of the `registers` command is `reg`.

#### Syntax

`reg[ regbank[ format]]`

where:

*regbank* Specifies the register bank to be listed. If you do not specify a register bank, the one named `Current` is listed. See *regbanks* on page 6-38 for details of how to specify a register bank.

*format* Specifies the format to be used in the list if you do not want the default format (see *format* on page 6-6).

#### Example

`reg user` Displays the number, name, and contents of each of the registers in the `user` register bank. You can issue a `regbk` command to see a list of the current register banks.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.52 reload

The `reload` command reloads the specified image. If you do not specify an image, the command reloads the current image. See *images* on page 6-30 for information on referring to images.

The shorthand form of the `reload` command is `rld`.

#### Syntax

`rld[ image]`

where:

*image* Specifies the image you want to reload.

#### Example

`rld` Reloads the current image. This can be useful if you have made changes to the image in memory and want to restore the image to its original state. To see the effect of this command, load an image, open a Memory processor view, and set its starting address to `0x8300`. Then in the CLI system view perform the examples given for the commands fillmem, savebinary, reload, and loadbinary.

### 6.6.53 run

The `run` command starts or restarts execution in the specified processor, or in the current processor if you do not specify a processor.

The shorthand form of the `run` command is `r`.

#### Alias

`go` is an alias for `run`.

#### Syntax

`r[ processor]`

where:

*processor* Specifies the processor (the current processor is the default).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.54 runmode

No longer a valid command. See *stepsize* on page 6-56.

### 6.6.55 runtopos

The `runtopos` command causes execution to proceed until the specified position is reached. The command applies to execution in the specified processor, or in the current processor if you do not specify one.

The shorthand form of the `runtopos` command is `rto`.

#### Syntax

`rto` *`position`*`[` *`processor`*`]`

where:

*`position`* Is an expression that evaluates to a memory address or line number.

*`processor`* Identifies the processor.

#### Example

`rto #1 | 130` Causes execution to run to position 130 in the file specified by index 1 in the file list.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.56 savebinary

The `savebinary` command copies the contents of the specified area of memory to the specified disk file.

The shorthand form of the `savebinary` command is `sb`.

#### Alias

`putfile` is an alias for `savebinary`.

#### Syntax

`sb` *`file expr1`* `[+]`*`expr2`*

where:

*`file`* Specifies the file in which you want to save the contents of the specified area of memory.

*`expr1`* Is an expression that evaluates to the starting address of the area of memory to save.

*`expr2`* Is an expression that evaluates either to the end address of the area of memory to save or, if preceded by `+`, to the number of bytes to save.

#### Example

```
sb sbtest.bin 0x8300 +256
```

Saves in a file called `sbtest.bin` the contents of the 256-byte area of memory starting at address `0x8300`. To see the effect of this command, load an image, open a Memory processor view, and set its starting address to `0x8300`, then in the CLI system view perform the examples given for the commands fillmem, savebinary, reload, and loadbinary.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.57 savesession

The `savesession` command saves details of the current debug session in a specified file. This allows the debug session to be restored to its current state at any future time. (To restore a debug session, see *loadsession* on page 6-34.)

The shorthand form of the `savesession` command is `ss`.

#### Syntax

`ss` *`file`*

where:

*`file`* Specifies the session file to be created.

### 6.6.58 setaci

The `setaci` command enables you to set the debugger internal `$aci_command` to a specified string. If the debugger internal `$aci_command` is not present then this command generates an error. There is no CLI interface to view the current setting of this property.

The shorthand form of the `setaci` command is `aci`.

#### Syntax

`aci` *`string`*


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.59 setbreakprops

The `setbreakprops` command enables you to set various properties of a breakpoint.

The breakpoint must already exist. Issue the command once for each breakpoint property to be set.

The shorthand form of the `setbreakprops` command is `sbp`.

#### Syntax

`sbp` *`breakpoint propid value`*

where:

*`breakpoint`* Identifies the breakpoint that is to have a property set.

*`propid`* Identifies the name of the property to be set as shown in Table 6-1.

**Table 6-1 Breakpoint properties**

| Property name | Type |
|---|---|
| state | Flag (enable or disable) |
| processor | Processor |
| condition | String or value |
| log_text | String |
| break_size | ASM |
| count | Integer |

Specify the name exactly as shown in the table, using lowercase characters.

*`value`* Specifies the setting you want the property to have. Each property takes its own type of setting as shown in the table.

#### Examples

```
sbp #3 state enable
```

Enables breakpoint number 3.

```
sbp #2 processor #1
```

Sets breakpoint number 2 to act on processor number 1.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.60 setimgprop

The `setimgprop` command sets an image-related internal variable to the specified value (see *imgproperties* on page 6-30). You need to supply either a string or an expression, depending on the type of the variable.

The shorthand form of the `setimgprop` command is `sip`.

#### Syntax

`sip` *`image ipvar value`*

where:

*`image`* Specifies the image that is to have an internal variable reset.

*`ipvar`* Specifies the ipvariable to be reset. See *Definitions* on page 6-9 for a list of valid ipvariables.

*`value`* Specifies the new value to be assigned to the specified variable.

#### Example

`sip` *`myimage`* `cmdline "-a -o -z"`

Specifies that whenever you load (or reload) the image *`myimage`*, it is supplied with the string `"-a -o -z"` as though you had entered that string after the image name on a command line. If the string consists of a single parameter, it does not need to be enclosed in quotes.

### 6.6.61 setmem

The `setmem` command sets the contents of memory at the specified address to the specified value.

The shorthand form of the `setmem` command is `smem`.

#### Syntax

`smem` *`addrexpr valexpr`*`[` *`memory`*`]`

where:

*`addrexpr`* Evaluates to the memory address at which you want to insert the new value.


Copyright © 1999-2001 ARM Limited. All rights reserved.

*valexpr* Evaluates to the value that you want to insert at the specified memory address. This evaluation results in an 8-bit, a 16-bit, or a 32-bit value depending on the setting of the memory parameter, or of the current global variable value if you do not specify the memory parameter.

*memory* If used must be set to 8, 16, or 32 (see *memory* on page 6-7).

#### Example

```
smem 0x83A8 0x41424344 32
```

Overwrites the 4 bytes of memory starting at address `0x83A8` with the 4-byte value `0x41424344`.

### 6.6.62 setpc

The `setpc` command sets the program counter to the specified value. The value you enter is evaluated according to the current setting of the input base variable.

The shorthand form of the `setpc` command is `pc`.

#### Syntax

`pc` *`expr`*

### 6.6.63 setproc

The `setproc` command makes the specified processor the current one. If other commands are issued with no processor specified, they apply to the current processor.

The shorthand form of the `setproc` command is `sproc`.

#### Syntax

`sproc` *`processor`*

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.64 setprocprop

The `setprocprop` command sets a processor-related internal variable to the specified value (see *procproperties* on page 6-37). You have to supply a value of the type required for the variable you are setting.

The shorthand form of the `setprocprop` command is `spp`.

#### Syntax

`spp` *`ppvariable value`* `[`*`processor`*`]`

where:

*`ppvariable`* Specifies the ppvariable to be reset. See *Definitions* on page 6-9 for a list of valid ppvariables.

*`value`* Specifies the new value to be assigned to the specified variable.

In the case of the `vector_catch` variable, you can supply a hexadecimal value, a decimal value, a string of characters, or a single character (see Table 6-2).

**Table 6-2 Allocation of bits in vector_catch variable**

| Bit | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Sets | Not used | F | I | Not used | D | P | S | U | R |

You are recommended to enclose a string of characters or a single character in quotes. If the string contains the character D or F, you must use quotes to avoid any attempt to interpret it as a hexadecimal value.

*`processor`* Specifies the ID number for the processor.

#### Examples

`spp vector_catch 0x00DF`

Is equivalent to using Processor Properties to check all the vector catch check boxes giving a setting of `RUSPDIF` instead of the default setting of `RUsPDif`. See *Configure Processor...* on page 5-96.

`spp vector_catch IuS`

Sets the I and S bits on, and the U bit off, leaving the other bits unchanged.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

`spp vector_catch "D"`

Sets the D bit on, and all the other bits unchanged.

`spp vector_catch 0`

Sets all the vector catch bits off. If you are debugging an embedded application with a JTAG-based debug agent like Multi-ICE, you can use this to free a watchpoint unit in the EmbeddedICE logic of the processor.

`spp semihosting_enabled 0`

Disables semihosting by not placing a breakpoint on the SWI vector. This also frees a watchpoint unit in the EmbeddedICE logic of the processor, for debugging an embedded application. For further information refer to the *ADS Debug Target Guide*.

### 6.6.65 setreg

The `setreg` command sets the specified register in the specified register bank to the value obtained by evaluating the specified expression (see *registers* on page 6-39). If you do not specify a register bank, the register bank named `Current` is used.

The expression can include a register bank name and register name.

If you are debugging an Angel target, you can set the registers of the current mode only.

The shorthand form of the `setreg` command is `sreg`.

#### Syntax

`sreg [`*`regbank`*`|]`*`register expr`*

#### Examples

`sreg r12 100` Sets register `r12` in register bank `current` to the value 100.

`sreg FIQ|r12 IRQ|r13`

Sets register `r12` in register bank `FIQ` to the value of register `r13` in register bank `IRQ`.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.66 setsourcedir

The `setsourcedir` command sets the list of paths to be searched.

The paths in this list, and the order in which they are listed, specify the paths searched when a source file is required. To display the current list, see *sourcedir* on page 6-53.

The shorthand form of the `setsourcedir` command is `ssd`.

#### Syntax

`ssd` *`dir_list`*`[` *`index`*`]`

where:

*`dir_list`* Specifies one or more fully-qualified directories, or is a null string.

*`index`* Specifies a position within the current list of directories.

The list is of fully qualified directory names. Enclose the list in quotation marks if it contains any spaces. If you are specifying multiple directories, separate them with `;` for Windows or `:` for UNIX.

To clear the current list, issue the `ssd` command with an empty string.

If you supply an index position, your directory list is inserted before the directory currently listed at the index number you specify.

If you do not supply an index position, your directory list overwrites any existing list of directories.

#### Examples

`ssd "c:\my srcs\proj A;d:\proj B;c:\srclib"`

Replaces any existing list.

`ssd ""` Clears the current list

`ssd c:\mydir2 #2`

Inserts the specified directory as the second in the list.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.67 setwatch

The `setwatch` command sets the specified expression to the specified value. This is of most use when the expression is one that is being watched (see *watch* on page 6-59).

The shorthand form of the `setwatch` command is `swat`.

#### Alias

`let` is an alias for `setwatch`.

#### Syntax

`swat` *`expr1 expr2`*

where:

| | |
|---|---|
| *`expr1`* | Specifies an expression to which you want to assign a value. |
| *`expr2`* | Specifies a new value to be assigned to the expression. |

#### Examples

| | |
|---|---|
| `swat a1 100` | Sets variable `a1` to the value 100. |
| `swat a b` | Sets variable `a` to the value of variable `b`. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.68 setwatchprops

The `setwatchprops` command enables you to set watchpoint properties. The watchpoint must already exist. Issue the command once for each watchpoint property to be set.

The shorthand form of the `setwatchprops` command is `swp`.

#### Syntax

`swp` *`watchpoint propid value`*

where:

*`watchpoint`* Identifies the watchpoint that is to have a property set.

*`propid`* Identifies the property to be set. You can identify a watchpoint property by its name, as shown in Table 6-3.

**Table 6-3 Watchpoint properties**

| Property name | Type |
|---|---|
| state | Flag (enable or disable) |
| processor | Processor |
| condition | String or value |
| log_text | String |
| value | Value |
| break_size | ASM |
| watch_size | Integer |
| count | Integer |

Specify the name exactly as shown in the table, using lowercase characters.

*`value`* Specifies the setting you want the property to have. Each property takes its own type of setting as shown in the table.

#### Examples

```
swp #3 state enable
```

Enables watchpoint number 3.


Copyright © 1999-2001 ARM Limited. All rights reserved.

`swp #2 value 17`

Set the value that is watched for by watchpoint number 2 to 17.

`swp #2 watch_size 32`

Force the size of watchpoint number 2 to 32 bits.

### 6.6.69 source

The `source` command displays the specified lines of the specified source file, in the following format:

```
linenumber<tab>sourcecode
```

The file must be associated with a loaded image.

The shorthand form of the `source` command is `src`.

#### Alias

`type` is an alias for `source`.

#### Syntax

`src` *`value1`* `[+]`*`value2`*`[` *`file`*`]`

where:

*`value1`* Specifies the line number in the source file at which you want the listing to begin.

*`value2`* Specifies either the last line number to be listed or, if preceded by `+`, the number of lines you want listed.

*`file`* Specifies the source file you want to list (by default the command lists the file associated with the current context).

#### Example

`src 111 +10` Displays lines 111 to 120 of source file `dhry_1.c` if you have the Dhrystone example program loaded and halted at the default breakpoint.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.70 sourcedir

The `sourcedir` command displays the list of paths searched when a source file is required, in the following format:

```
index<tab>fully qualified directory name
```

The paths, on local or remote machines, are searched in the listed order.

To change the list of paths, see *setsourcedir* on page 6-49.

The shorthand form of the `sourcedir` command is `sdir`.

#### Syntax

```
sdir
```

### 6.6.71 stackentries

The `stackentries` command lists the current backtrace information stored in the debugger describing the current execution context. Each stack entry is listed on a separate line, in the following format:

```
index<tab>stackentry
```

The index value given in this list enables you to specify individual stack entries in other commands. See *convariables* on page 6-21 and *context* on page 6-20, for example.

The shorthand form of the `stackentries` command is `stk`.

#### Alias

`backtrace` is an alias for `stackentries`.

#### Syntax

`stk[ count]`

where:

*count* Specifies the number of lines you want listed if you do not want the whole stack displayed.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.72 stackin

The `stackin` command sets the current context to that of the called procedure or method.

The shorthand form of the `stackin` command is `in`.

#### Syntax

```
in
```

### 6.6.73 stackout

The `stackout` command sets the current context to that of the calling procedure or method.

The shorthand form of the `stackout` command is `out`.

#### Syntax

```
out
```

### 6.6.74 statistics

The `statistics` command adds a new reference point with a specified name or displays all the current reference points, in the following format:

```
reference point name<tab>value
```

The shorthand form of the `statistics` command is `stat`.

#### Syntax

`stat[ ref_pt_name]`

where:

*ref_pt_name* Specifies the name of a new reference point that you want to add. If the name already exists, an error message is displayed.

If you do not supply a reference point name, the statistics at all reference points are displayed.

See also *clearstat* on page 6-18 for a description of how to delete the statistics at a specified reference point.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.75 step

The `step` command causes execution to proceed by one step.

The `step` command is not available when executing Jazelle instructions. Submitting the command when the processor is in Jazelle state produces an error.

The `step` and `step in` commands are not available where the current instruction is a BXJ instruction.

The shorthand form of the `step` command is `st`.

#### Syntax

`st[ step][ instr]`

where:

*`step`* Can be set to `in` or `out` (see *step* on page 6-7).

*`instr`* Can be set to `line` or `instr` (see *instr* on page 6-7).

#### Examples

`step in line` Steps one source line. If the line contains a subroutine call, steps into the subroutine.

`step out instr`

Steps out of the current stack. If no stack frame information is available, steps one instruction.

`step` Steps, without forcing a step in or out, one instruction or source line depending on the setting of `instr` (see *stepsize* on page 6-56). If a subroutine call is encountered, this command steps over it.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.76 stepsize

The `stepsize` command enables you to examine or set the step size. To see the current setting, issue the command with no parameter.

The shorthand form of the `stepsize` command is `ssize`.

#### Syntax

`ssize[ instr]`

where:

*instr* Can be set to `instr` or `line` (see *instr* on page 6-7), but is overridden if no source is available.

### 6.6.77 stop

The `stop` command stops execution of the specified processor, or of the current processor if you supply no parameter.

There is no shorthand form of the `stop` command.

#### Syntax

`stop[ processor]`

### 6.6.78 trace

The `trace` command toggles the trace status on or off. This command is effective only when the add-on product *Trace Debug Tools* (TDT) is licensed for use and the target supports trace.

There is no shorthand form of the `trace` command.

#### Syntax

`trace on|off`

The command displays no output if it succeeds. If unsuccessful, it displays one of the following error messages:

- `not a valid trace target`
- `fail to start/stop trace`

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.79 traceload

The `traceload` command loads a Trace configuration file. This command is effective only when the add-on product *Trace Debug Tools* (TDT) is licensed for use and the target supports trace.

The shorthand form of the `traceload` command is `trload`.

#### Syntax

`trload` *`tcfile`*

where:

*`tcfile`* Specifies the Trace configuration file to be read.

### 6.6.80 type

See *source* on page 6-52.

### 6.6.81 unbreak

See *clearbreak* on page 6-17.

### 6.6.82 update

The `update` command enables you to specify whether or not screen updates take place while commands from an Obey file are being executed.

The shorthand form of the `update` command is `upd`.

#### Syntax

`upd` *`toggle`*

where:

*`toggle`* Can be set to either `on` or `off`.

For further information see *Command Line Interface Properties dialog, General tab* on page 5-67.

### 6.6.83 unwatch

See *clearwatch* on page 6-19.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 6.6.84 variables

The `variables` command lists all the global variables of the specified image, or of the current image if you do not specify an image. Each variable is listed on a separate line, in the following format:

```
index<tab>varname
```

The position in this list, `index`, gives you a convenient way of referring to a variable.

The shorthand form of the `variables` command is `va`.

#### Syntax

`va[ image]`


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 6.6.85 watch

The `watch` command displays the name, type, and value of the specified expression, in the following format:

```
name<tab>type<tab>value
```

The command displays a simple expression according to the specified format, or the default format if you do not specify one (see also *format* on page 6-6). It displays a complex expression after suitably expanding it. See also *setwatch* on page 6-50.

The shorthand form of the `watch` command is `wat`.

#### Alias

`print` is an alias for `watch`.

#### Syntax

`wat` *`expr`*`[` *`format`*`]`

#### Example

```
wat 5*Int_1_Loc-Int_2_Loc==10 dec
```

Displays the value `1` (true) if `5*Int_1_Loc-Int_2_Loc` evaluates to `10`, or `0` (false) otherwise.

### 6.6.86 watchpt

If you supply parameters, the `watchpt` command creates and sets a new watchpoint so that execution continues normally until the value stored at the specified location changes for the *n*th time. If you do not specify *n* it takes a default value of 1.

If you supply no parameters, the `watchpt` command lists all the watchpoints that are currently set. Each watchpoint is shown on a separate line, after a first line containing the headings for the following columns:

`Index` The position in the list. This gives you a convenient way of referring to a watchpoint in commands such as `clearwatch`.

An X is displayed if the watchpoint is currently disabled.

`Item` This shows the fully-qualified source code filename in brackets, a colon, and the source code line number at which the watchpoint is set. It also shows, in square brackets, the corresponding memory address, and, if set, the value being watched for.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

`Watching` This describes what you are watching.

`Size` This shows whether the watchpoint is ARM-sized (`4bytes`), Thumb-sized (`2bytes`), or a single byte (`1byte`).

`Count` Two numbers are shown. The first is the number of times the value stored at the watchpoint has changed since the last time the watchpoint was triggered. The second shows the number of times the value has to change to trigger the watchpoint.

`Condition` Any condition that you have specified that must also be satisfied before the watchpoint can be triggered is shown here.

`Additional` The final column displays additional information. This can include one or more of the following:

- `S/HW` Shows whether the watchpoint is implemented in hardware or software.
- `(ID)` A hardware watchpoint can have a hardware resource identifier. Any such identifier is shown here.
- `Processor` Identifies the processor in which the watchpoint is set.
- `Action` This shows whether the action taken when the watchpoint is triggered is to stop execution of the target (`Break`) or to log the event (`Log`).

The shorthand form of the `watchpt` command is `wpt`.

### Syntax

`wpt[ expr[ n]]`

### Example

`wpt 0x83A8 5` Sets a watchpoint at address `0x83A8`, requiring 5 changes of content to trigger it.

When you have created a new watchpoint, you can change its properties with the `setwatchprops` command. See *setwatchprops* on page 6-51.

Bitfields are not watchable.

If you are debugging through JTAG or EmbeddedICE logic, ensure that watchpoints on global or static variables use hardware watchpoints to avoid any performance penalty.

You can also use the `wpt` command to set an AXD watchpoint on a range of addresses. For example:

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

```
wpt (char[16])*0xF200
```

traps all data changes that take place in the 16 bytes of memory starting at `0xF200`.

For this to work efficiently when you are debugging with, for example, Multi-ICE, ensure that the size of the watchpoint in bytes is a power of 2, and that the address of the watchpoint is aligned on a size-byte boundary. Accesses to the area you specify are trapped only if they change any value stored there. A replacement of a value with the same value, for example, is not trapped.

### 6.6.87 where

The `where` command displays information about the specified context, or about the current context if you do not supply a parameter. The command displays the source file name, line number, and source line if the source is available. Otherwise the command displays the disassembled instruction (see *stackentries* on page 6-53).

There is no shorthand form of the `where` command.

#### Syntax

```
where[ context]
```


Copyright © 1999-2001 ARM Limited. All rights reserved.

Copyright © 1999-2001 ARM Limited. All rights reserved.

# Part B

## armsd

# Chapter 7
# About armsd

The *ARM Symbolic Debugger* (armsd) is an interactive source-level debugger that provides debugging support for languages such as C, and low-level support for ARM assembly language. It is a command-line debugger that runs on all supported platforms. This chapter contains the following sections:


- *About armsd* on page 7-2
- *Command syntax* on page 7-3.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 7.1 About armsd

The *ARM symbolic debugger* (armsd) can be used to debug programs built using the ARM tools.

### 7.1.1 Selecting a debugger

armsd supports:

- debugging using ARMulator
- remote debugging using ADP.

### 7.1.2 Automatic command execution on startup

You normally enter armsd commands from the keyboard, or by specifying a script file containing commands, but before armsd accepts any of this input it obeys commands from an initialization file, if one exists.

The initialization file is called `armsd.ini`. The current directory is searched first for this file, then the directory specified by the environment variable `ARMHOME`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 7.2 Command syntax

You invoke armsd using the command given below. Underlining shows the permitted abbreviations.

The full list of commands available when armsd is running is given in *Alphabetical list of armsd commands* on page 9-7.

### 7.2.1 Command-line options

```
armsd [-help] [-vsn] [[-little]|[-big]] [-cpu name] [-fpe|[-nofpe]] [-symbols]
[-o name] [-script name] [-exec] [-i name] [-clock n] [-target dllname]
[[-remote]|[-armul]|[-adp options]] image_name args
```

where:

`-help`
: Gives a summary of the armsd command-line options.

`-vsn`
: Displays information on the armsd version.

`-little`
: Specifies that memory is to be little-endian (the default setting).

`-big`
: Specifies that memory is to be big-endian.

`-cpu` *`name`*
: Specifies the CPU type that is to be simulated. With this option you must not specify `-rem` or `-adp` as the target. Specify `-armul` as the target to invoke ARMulator. If you do not specify a target, ARMulator is invoked if it can simulate the specified processor. If the specified processor cannot be simulated, `armsd` exits. Instead of *`name`* you can specify `list`, to display a list of processors available on the target. For example:

  `armsd -cpu list`
  : lists available processors of standard targets (ARMulator and Remote_A)

  `armsd -armul -cpu list`
  : lists available processors of ARMulator

  `armsd -target` *`dllname`* `-cpu list`
  : lists available processors of the specified target.

  ARMulator is the only ARM supplied target that has a list of available processors.

  Instead of `-cpu` you can still use `-proc`, but this is now obsolete.

`-fpe`
: Instructs ARMulator to load the FPE on startup.


Copyright © 1999-2001 ARM Limited. All rights reserved.

`-nofpe` Instructs ARMulator not to load the FPE on startup (this is the default setting).

`-symbols` Reads debug information from the specified image file but does not download the image.

`-o` *name* Writes output from the debuggee to the named file.

`-script` *name* Takes commands from the named file (reverts to `stdin` on reaching EOF).

`-exec` Instructs the debugger to load and execute the named file immediately, and quit when execution stops.

`-i` *name* Adds *name* to the set of paths to be searched to find source files.

`-clock` *n* Specifies the clock speed in Hz (suffixed with K or M) for ARMulator. This is intended for use with an `armsd.map` file.

`-target` *dllname* Specifies a `.dll` file that is a third-party RDI target simulator to be used instead of ARMulator.

`-remote` Selects remote debugging. By default this is ADP.

`-armul` Selects ARMulator. This is assumed by default if you do not specify a target but do specify a processor type that ARMulator can simulate.

`-adp` *options* Selects remote debugging using ADP, further defined by one or more of the following options:

`-port` *expr*

specifies the ADP port to use, where *expr* selects serial, serial-and-parallel, or ethernet communications and can be one of:

`s=`*n* Selects serial port communications. *n* can be 1, 2 or a device name.

`s=`*n*`,p=`*m* Selects serial-and-parallel port communication. *n* and *m* can be 1, 2, or a device name. There must be no space between the arguments.

`e=`*id* Selects ethernet communication. *id* is the ethernet address of the target board.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

For serial and serial-and-parallel communications, you can add `,h=0` to the port expression to switch off the heartbeat feature of ADP. For example, `-port s=1,h=0` selects serial port 1 and turns off the ADP heartbeat.

`-linespeed n`

Sets the line speed to *n*.

`-loadconfig name`

Specifies a file containing required configuration data, when using a Remote_A connection to EmbeddedICE. See *loadconfig* on page 9-47 for more information.

`-selectconfig name version`

Specifies the target for which configuration data is required, when using a Remote_A connection to EmbeddedICE. See *selectconfig* on page 9-48 for more information.

*image_name*

Gives the name of the file to debug. You can also specify this information using the load command. See *load* on page 9-28 for more information.

*args*

Gives program arguments. You can also specify this information using the `load` command. See *load* on page 9-28 for more information.

**Note**

Where given, the debug file, and any associated arguments, must be the last entry on the command line. This ensures that all specified command-line options are correctly interpreted.


Copyright © 1999-2001 ARM Limited. All rights reserved.

Copyright © 1999-2001 ARM Limited. All rights reserved.

# Chapter 8
# Getting Started in armsd

This chapter includes further information about the use of the *ARM Symbolic Debugger* (armsd). It contains the following sections:


- *Specifying source-level objects* on page 8-2
- *armsd variables* on page 8-7
- *Low-level debugging* on page 8-13.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 8.1 Specifying source-level objects

This section gives information on syntax conventions, variables, program locations, expressions, and constants. It contains the following subsections:

- *Command syntax conventions*
- *Variable names and context*
- *Program locations* on page 8-4
- *Expressions* on page 8-5
- *Constants* on page 8-6.

### 8.1.1 Command syntax conventions

The following conventions are used in descriptions of armsd commands:

`monospace` Shows command elements that you should type at the keyboard.

`monospace` Many command names can be abbreviated. Underlined text shows the permitted abbreviation of a command.

*`monospace`* Represents an item such as a filename or variable name. Replace this with the name of your file, variable, and so on.

`{}` Items in braces are optional. The braces are for clarity. Do not type them. In the one case where braces are required by the debugger, these are enclosed in quotation marks in the syntax pattern.

`*` A star (`*`) following a set of braces means that the items in those braces can be repeated as many times as required.

### 8.1.2 Variable names and context

You can usually just refer to variables by their names in the original source code. To print the value of a variable, type:

```
print variable
```

#### High-level languages

With structured high-level languages, you can access variables defined in the current context by giving their names. Other variables must be preceded by the context (for example, the name of the function) in which they are defined. This also gives access to variables that are not visible to the executing program at the point at which they are being examined. The syntax is:

```
procedure:variable
```

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Global variables

You can access global variables by qualifying them with themodule name or filename if there is any ambiguity. For example, because the module name is the same as a procedure name, you must prefix the filename or module name with #. The syntax is:

```
#module:variable
```

### Ambiguous declarations

If a variable is declared more than once within the same procedure,resolve the ambiguity by qualifying the reference with the line number in which the variable is declared as well as, or instead of, the function name:

```
#module:procedure:line-no:variable
```

### Variables within activations of a function

To pick out a particular activation of a repeated or recursive function call, prefix the variable name with a backslash (\) followed by an integer. Use 1 for the first activation, 2 for the second, and so on. A negative number looks backwards through activations of the function, starting with \-1 for the previous one. If no number is specified and multiple activations of a function are present, the debugger always looks at the most recent activation.

To refer to a variable within a particular activation of a function, use:

```
procedure\{-}activation-number:variable
```

#### *Expressing context*

The complete syntax for the various ways of expressing contextis:

```
{#}module{{:procedure}*
{\{-}activation-number}}
{#}procedure{{:procedure}*
{\{-}activation-number}}#
```

#### *Specifying variable names*

The complete syntax for specifying a variable name is:

```
{context:.{line-number::}}variable
```

The various syntax extensions required to differentiate between objects rarely have to be used together.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 8.1.3 Program locations

Some commands require arguments that refer to locations in the program. You can refer to a location in the program by:

- procedure name
- program line number
- statement within a line.

In addition to the high-level program locations described here, you can also specify low-level locations. See *Entering addresses* on page 4-12 and *Low-level symbols* on page 8-13 for further details.

#### Procedure name

Using a procedure name alone sets a breakpoint (see *break* on page 9-12) at the entry point of that procedure.

#### Program line number

Program line numbers can be qualified in the same way as variable names, for example:

```
#module:123procedure:3
```

Line numbers can sometimes be ambiguous, for example when a file is included within a function. To resolve any ambiguities, add the name of the file or module in which the line occurs in parentheses. The syntax is:

```
number(filename)
```

#### Statement within a line

To refer to a statement within a line, use the line number followed by the number of the statement within the line, in the form:

```
line-number.statement-number
```

So, for example, `100.3` refers to the thirdstatement in line 100.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 8.1.4 Expressions

Some debugger commands require expressions as arguments. Their syntax is based on C. A full set of operators is available. Lower-numbered operators have higher precedence. These are shown in Table 8-1, in descending order of precedence.

**Table 8-1 Precedence of operators**

| Precedence | Operator | Purpose | Syntax |
|---|---|---|---|
| 1 | `()` | Grouping | `a * (b + c)` |
| | `[]` | Subscript | `isprime[n]` |
| | `.` | Record selection | `rec.field,a.b.c` |
| `rec->next` | `->` | Indirect selection | `rec->next` is identical to `(*rec).next` |
| 2 | `!` | Logical NOT | `!finished` |
| | `~` | Bitwise NOT | `~mask` |
| | `-` | Unary minus | `-a` |
| | `*` | Indirection | `*ptr` |
| | `&` | Address | `&var` |
| 3 | `*` | Multiplication | `a * b` |
| | `/` | Division | `a / b` |
| | `%` | Integer remainder | `a % b` |
| 4 | `+` | Addition | `a + b` |
| | `-` | Subtraction | `a - b` |
| 5 | `>>` | Right shift | `a >> 2` |
| | `<<` | Left shift | `a >> 2` |
| 6 | `<` | Less than | `a < b` |
| | `>` | Greater than | `a > b` |
| | `<=` | Less than or equal | `a <= b` |
| | `>=` | Greater than or equal | `a >= b` |
| 7 | `==` | Equal | `a == 0` |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table 8-1 Precedence of operators (continued)**

| Precedence | Operator | Purpose | Syntax |
|---|---|---|---|
| | `!=` | Not equal | `a != 0` |
| 8 | `&` | Bitwise AND | `a & b` |
| 9 | `^` | Bitwise EOR | `a ^ b` |
| 10 | `\|` | Bitwise OR | `a \| b` |
| 11 | `&&` | Logical AND | `a && b` |
| 12 | `\|\|` | Logical OR | `a \|\| b` |

You can only apply subscripting to pointers and array names. The symbolic debugger checks both the number of subscripts and their bounds, in languages that support this checking. You are advised not to use out-of-bound array accesses. As in C, you can use the name of an array without subscripting to yield the address of the first element.

Use the prefix indirection operator `*` to dereference pointer values. If `ptr` is a pointer, `*ptr` yields the object to which it points.

If the left-hand operand of a right shift is a signed variable, the shift is an arithmetic one and the sign bit is preserved. If the operand is unsigned, the shift is a logical one and zero is shifted into the most significant bit.

**Note**

Expressions must not contain function calls that return nonprimitive values.

### 8.1.5 Constants

Constants can be decimal integers, floating-point numbers, octal integers, or hexadecimal integers. The constant `1` is an integer whereas `1.` is a floating-point number.

Character constants are also allowed. For example, `A` yields 65, the ASCII code for the character A.

You can specify address constants by the address preceded with an `@` symbol. For commands that accept low-level symbols by default, you can omit the `@`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 8.2 armsd variables

This section lists the variables available in armsd, and gives information on manipulating them. It contains the following subsections:

- *Summary of armsd variables*
- *Accessing variables* on page 8-10
- *Formatting printed results* on page 8-11
- *Specifying the base for input of integer constants* on page 8-12.

### 8.2.1 Summary of armsd variables

You can modify many debugger defaults by setting variables. Table 8-2 lists the variables. Most of these are described elsewhere in this chapter in more detail.

**Table 8-2 armsd variables**

| Variable | Description |
|---|---|
| `$clock` (ARMulator only) | Number of microseconds since simulation started. This read-only variable is available only if a processor clock speed is specified. See *ARMulator configuration* on page 5-88 for information on specifying the simulated processor clock speed. |
| `$cmdline` | Argument string for the debuggee. |
| `$echo` | Nonzero to echo commands from obeyed files (initially 1). |
| `$examine_lines` | Default number of lines for `examine` command (initially 8). |
| `$int_format` | Default format for printing integer values (initially “0x%.8lx”). |
| `$float_format` | Default format for printing floating-point values (initially “%g”). |
| `$uint_format` | Default format for printing unsigned integer values (initially “0x%.8lx”). |
| `$sbyte_format` | Default format for printing signed byte values (initially “%c”). |
| `$ubyte_format` | Default format for printing unsigned byte values (initially “%c”). |
| `$string_format` | Default format for printing string values (initially “%s”). |
| `$complex_format` | Default format for printing complex values (initially “(%g,%g)”). |
| `$pointer_format` | Default format for printing pointer values (initially “0x%.8lx”). |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table 8-2 armsd variables (continued)**

| Variable | Description |
|---|---|
| `$inputbase` | Base for input of integer constants (initially 10). |
| `$list_lines` | Default number of lines for list command (initially 16). |
| `$fpresult` | Floating-point value returned by last called function (junk if none, or if a floating-point value was not returned). A read-only variable. `$fpresult` returns a result only if the image has been built for hardware floating-point. If the image is built for software floating-point, it returns zero. |
| `$type_lines` | Default number of lines for the `type` command. |
| `$memory_statistics` (ARMulator only) | Outputs any memory map statistics that ARMulator has been keeping. A read-only variable. See *ARMulator configuration* on page 5-88 for further details. |
| `$statistics` (ARMulator only) | Outputs any statistics which ARMulator has been keeping. A read-only variable. |
| `$statistics_inc` (ARMulator only) | Similar to `$statistics`, but outputs the difference between the current statistics and those when `$statistics` was last read. A read-only variable. |
| `$vector_catch` | Indicates whether or not execution is interrupted when various exceptions occur. The default value is `%RUsPDAifE`. Capital letters indicate that the exception is to be intercepted:<br>`R` Reset<br>`U` Undefined Instruction<br>`S` SWI<br>`P` Prefetch Abort<br>`D` Data Abort<br>`A` Reserved (do not use)<br>`I` IRQ<br>`F` FIQ<br>`E` Reserved (do not use) |
| `$rdi_log` | RDI logging is enabled if nonzero, and serial line logging is enabled if bit 1 is set (initially 0). |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table 8-2 armsd variables (continued)**

| Variable | Description |
|---|---|
| `$top_of_memory` | This variable informs the debugger where the top of RAM is on your target.<br>This is used to enable Multi-ICE, EmbeddedICE, and Angel to return sensible values when a `HEAP_INFO SWI` call is made to determine where to place the heap and stack in memory. The default is `0x80000` (512KB). Modify this before executing a program on the target if the memory available differs from this. |
| `$target_fpu` | This variable controls the way that floating-point values are interpreted by the debugger. It is important for correct display of float and double values in memory that this variable is set to a value that is appropriate for the target in use. If you attempt to change this value, a validity test ensures that the only settings allowed are those that are compatible with the representation of floating-point values in the current image. Valid settings and their meanings are:<br>**1** Selects pure-endian doubles (`softVFP`). This is the default setting for images built with ADS tools. Values are read from ordinary registers.<br>**2** Selects mixed-endian doubles (`softFPA`). Values are read from ordinary registers.<br>**3** Selects hardware Vector Floating-Point unit (`VFP`). Values are read from registers CP10 and CP11.<br>**4** Selects hardware Floating-Point Accelerator (`FPA`). Values are read from registers CP1 and CP2.<br>**5** Reserved.<br>Incompatible settings are accepted but a warning is given.<br>SoftVFP and SoftFPA images run correctly on a target whether or not hardware floating point is present. FPA images can also run correctly without hardware floating point, but only if the Floating Point Emulator in ARMulator is active. VFP images require appropriate hardware. For further information, see the *ADS Compilers and Libraries Guide*. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table 8-2 armsd variables (continued)**

| Variable | Description |
|---|---|
| `$sourcedir` | This variable contains a list of the paths to be searched when a source file is required. It defaults to NULL if no value is specified. When you specify search paths:<br>• Enclose the full pathname in double quotes.<br>• In armsd under Windows DOS, escape the backslash directory separator with another backslash character. For example:<br>`$sourcedir="c:\\mysource\\src1"`<br>• Separate multiple pathnames with a semicolon, not with a space character. For example:<br>`$sourcedir="c:\\my src\\src1;c:\\my src\\src2"` |
| `$result` | Integer result returned by last called function (junk if none, or if an integer result was not returned). A read-only variable. |
| `$semihosting_enabled` | Enables or disables semihosting (see *Definitions* on page 6-9). |

### armsd internal variables

The variables in Table 8-3 are included to support EmbeddedICE.

**Table 8-3 armsd variables for Multi-ICE and EmbeddedICE**

| Variable | Description |
|---|---|
| `$icebreaker_lockedpoints` | Shows or sets locked EmbeddedICE logic points. |
| `$semihosting_vector` | Sets up semihosting SWI vector (described in the *ADS Debug Target Guide*). |
| `$semihosting_arm_swi` | Defines which ARM SWIs are interpreted as semihosting requests by the debug agent. The default is `0x123456`. Do not change this. |
| `$semihosting_thumb_swi` | Defines which Thumb SWIs are interpreted as semihosting requests by the debug agent. The default is `0xAB`. Do not change this. |

## 8.2.2 Accessing variables

The following commands are available for accessing variables.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### print

This command examines the contents of variables in the debugged program, or displays the result of arbitrary calculations involving variables and constants. Its syntax is:

```
p{rint}{/format} expression
```

For example:

```
print/%x listp->next
```

prints field `next` of structure `listp`.

If no format string is entered, integer values default to the format described by the variable `$int_format`. The default format string for floating-point values is `%g`. By default, pointer values are printed in hexadecimal notation using the format string `0x%.8lx`, for example, `0x000100E4`.

### let

The `let` command enables you to change the value of a variable or contents of a memory location. Its syntax is:

```
{let} variable = expression{{,} expression}*
{let} memory-location = expression{{,} expression}*
```

You can use an equals sign (=) or a colon (:) to separate the variable or location from the expression. If you specify multiple expressions, separate them by commas or spaces.

You can change variables to compatible types of expression only. However, the debugger performs conversions between integer and floating-point values if necessary, rounding to zero. You can change the value of an array, but not its address, because array names are constants. If you omit the subscript, it defaults to zero. If you specify multiple expressions, each expression is assigned to `variable[`*n*`-1]`, where *n* is the nth expression.

The `let` command is used in low-level debugging to change memory. If the left-hand expression is a constant or a true expression (and not a variable) it is treated as a word address, and memory at that location (and if necessary the following locations) is changed to the values in the following expression(s).

## 8.2.3 Formatting printed results

You can set the default format strings used by the `print` command for the output of results of various types of data by using `let` with the following variable names:

- `$int_format`
- `$uint_format`
- `$float_format`


Copyright © 1999-2001 ARM Limited. All rights reserved.

- `$sbyte_format`
- `$ubyte_format`
- `$string_format`
- `$complex_format`
- `$pointer_format`.

For example, you can change the value of the root-level variable `$int_format` from its initial setting of `"0x%.81x"` to another value with a command of the form:

```
{let} $int_format = string
```

The initial value of each of these format variables is given in *Summary of armsd variables* on page 8-7.

### 8.2.4 Specifying the base for input of integer constants

You use the `$inputbase` variable to set the base used for the input of integer constants.

```
{let} $inputbase = expression
```

If the input base is set to 0, numbers are interpreted as octal if they begin with 0. Regardless of the setting of `$inputbase`, hexadecimal constants are recognized if they begin with `0x`.

**Note**

`$inputbase` only specifies the base for the input of numbers. For information on output formats see *Formatting printed results* on page 8-11.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## 8.3 Low-level debugging

Low-level debugging tables are generated automatically during linking (unless linked with `-nodebug`).

There is no need to enable debugging at the compilation stage for low-level debugging only.

### 8.3.1 Low-level symbols

Low-level symbols are differentiated from high-level ones by preceding them with `@`.

The differences between high and low-level symbols are:

- a low-level symbol for a procedure refers to its call address, often the first instruction of the stack frame initialization
- the corresponding high-level symbol refers to the address of the code generated by the first statement in the procedure.

You can use low-level symbols with most debugger commands. For example, when used with the `watch` command they stop execution if the word at the location named by the symbol changes. You can also use low-level symbols where a command expects an address expression.

Certain commands (`list`, `find`, `examine`, `putfile`, and `getfile`) accept low-level symbols by default. To specify a high-level symbol, precede it by `^`.

You can also use memory addresses with commands. These must also be preceded by `@`. For further details see *Entering addresses* on page 4-12.

**Note**

Low-level symbols do not have a context and so they are always available.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 8.3.2 Predefined symbols

There are several predefined symbols, as shown in Table 8-4. To differentiate these from any high-level or low level symbols in the debugging tables, precede them with #.

**Table 8-4 High-level symbols for low-level entities**

| Symbol | Description |
|---|---|
| r0 to r14 | The general-purpose ARM registers 0 to 14. |
| r15 | The address of the instruction that is about to execute. This can include the condition code flags, interrupt enable flags, and processor mode bits, depending on the target ARM architecture. This value can be different from the real value of register 15 due to the effect of pipelining. |
| pc | The address of the instruction that is about to execute. |
| sp | The stack pointer (r13). |
| lr | The link register (r14) |
| fp | The frame pointer (r11). |
| psr and cpsr | psr and cpsr are synonyms for the program status register for the current mode. The values displayed for the condition code flags, interrupt enable flags, and processor mode bits, are an alphabetic letter for each condition code and interrupt enable flag, and a mode name (preceded by an underscore) for the mode bits. The mode name is one of USER, IRQ, FIQ, SVC, UNDEF, ABORT, and SYSTEM. Mode values out of normal ranges are labeled Reserved_nn. 26-bit mode is no longer supported by the ARM tool chain. |
| spsr | spsr is the saved program status register for the current mode. The values displayed are listed above in psr and cpsr. |
| f0 to f7 | The FPA floating-point registers 0 to 7. |
| fpsr | The FPA floating-point status register. |
| fpcr | The FPA floating-point control register. |
| a1 to a4 | These are ATPCS register names. They refer to arguments 1 to 4 in a procedure call (stored in r0 to r3). |
| v1 to v7 | These are ATPCS register names. They refer to the 5, 6, or 7 general-purpose register variables that the compiler allocates (stored in r4 to r10). |
| sb | This is the ATPCS static base in RWPI variants of the ATPCS (r9/v6). |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table 8-4 High-level symbols for low-level entities (continued)**

| Symbol | Description |
|---|---|
| `sl` | This is the ATPCS stack limit register, used in variants of the ATPCS that implement software stack limit checking (r10/v7). |
| `ip` | This is the ATPCS intra-procedure scratch register, used in procedure entry and exit and as a scratch register (r12). |
| `s0` to `s31` | VFP single-precision data registers. Applicable only to targets with a VFP coprocessor. |
| `d0` to `d15` | VFP double-precision data registers. Applicable only to targets with a VFP coprocessor. |

### Printing register information

You can examine all these registers with the `print` command and change them with the `let` command. For example, the following form displays the *Program Status Register* (PSR):

```
print/%x #psr
```

### Setting the PSR

The `let` command can also set the PSR, using the usual syntax for PSR flags.

For example, you can set the `N` and `F` flags, clear the `V` flag, and leave the `I`, `Z`, and `C` flags untouched and the processor set to 32-bit supervisor mode, by typing:

```
let #psr = %NvF_SVC
```

The following example changes to User mode:

```
psr = %_User
```

**Note**

The percentage sign must precede the condition flags and the underscore which in turn must precede the processor mode description.

### Using # with low-level symbols

Normally, you do not have to use # to access a low-level symbol. You can use # to force a reference to a root context if you see the error message:

```
Error: Name not found
```


Copyright © 1999-2001 ARM Limited. All rights reserved.

For example, use `#pc=0` instead of `pc=0`.


Copyright © 1999-2001 ARM Limited. All rights reserved.

# Chapter 9
# Working with armsd

This chapter lists and explains every command supported by the *ARM Symbolic Debugger* (armsd). It contains the following sections:


- *Groups of armsd commands* on page 9-2
- *Alphabetical list of armsd commands* on page 9-7
- *Accessing the debug communications channel* on page 9-46
- *armsd commands for EmbeddedICE* on page 9-47.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## 9.1 Groups of armsd commands

This section lists all armsd commands in functional groups. The commands are explained individually in *Alphabetical list of armsd commands* on page 9-7.

The functional groups are:

- *Symbols*
- *Controlling execution*
- *Reading and writing memory* on page 9-3
- *Program context* on page 9-3
- *Low-level debugging* on page 9-3
- *Coprocessor support* on page 9-4
- *Profiling commands* on page 9-5
- *Miscellaneous commands* on page 9-5.

The semicolon character (;) separates two commands on a single line.

**Note**

The debugger queues commands in the order it receives them, so that any commands attached to a breakpoint are not executed until all previously queued commands have been executed.

### 9.1.1 Symbols

These commands allow you to view information on armsd symbols:

`symbols` Lists all symbols, such as variables and function names, defined in the given or current context, along with their type information.

`variable` Provides type and context information on the specified variable (or structure field).

`arguments` Shows the arguments that were passed to the current procedure, or another active procedure.

### 9.1.2 Controlling execution

These commands allow you to control execution of programs by setting and clearing watchpoints and breakpoints, and by stepping through instructions and statements:

`break` Adds breakpoints.

`call` Calls a procedure.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

`go` Starts execution of a program.

`istep` Steps through one or more instructions.

`load` Loads an image for debugging.

`reload` Reloads the object file specified on the armsd command line, or with the last `load` command.

`step` Steps execution through one or more statements.

`unbreak` Removes a breakpoint.

`unwatch` Clears a watchpoint.

`watch` Adds a watchpoint.

### 9.1.3 Reading and writing memory

These commands allow you to set and examine program context:

`getfile` Reads from a file and writes the content to memory.

`putfile` Writes the contents of an area of memory to a file.

### 9.1.4 Program context

These commands allow you to set and examine program context:

`where` Prints the current context as a procedure name, line number in the file, filename and the line of code.

`backtrace` Prints information about all currently active procedures.

`context` Sets the context in which the variable lookup occurs.

`out` Sets the context to be the same as that of the caller of the current context.

`in` Sets the context to that called from the current level.

### 9.1.5 Low-level debugging

These commands allow you to select low-level debugging and to display the contents of memory, registers, and low-level symbols:

`language` Sets up low-level debugging if you are already using high-level debugging.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

`registers` Displays the contents of ARM registers 0 to 14, the PC and the status flags contained in the PSR.

`fpregisters` Displays the contents of the eight floating-point registers f0 to f7 and the floating-point program status register FPSR.

`examine` Allows you to examine the contents of the memory between a pair of addresses, displaying it in both hexadecimal and ASCII formats, with 16 bytes per line.

`list` Displays the contents of the memory between a specified pair of addresses in hexadecimal, ASCII and instruction format, with four bytes (one instruction) per line.

`find` Finds all occurrences in memory of a given integer value or character string.

`lsym` Displays low-level symbols and their values.

### 9.1.6 Coprocessor support

The symbolic debugger includes coprocessor support that enables access to registers of a coprocessor through a debug monitor that is ignorant of the coprocessor. This is only possible if the registers of the coprocessor are read (if readable) and written (if writable) by a single `MRC`, `MCR`, `LDC` or `STC` instruction in a non-User mode. For coprocessors with more unusual registers, there must be support code in a debug monitor. The following commands are available:

`coproc` Describes the register set of a coprocessor and specifies how the contents of the registers are formatted for display.

`cregdef` Describes how the contents of a coprocessor register are formatted for display.

`cregisters` Displays the contents of all readable registers of a coprocessor, in the format specified by an earlier `coproc` command.

`cwrite` Writes to a coprocessor register.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 9.1.7 Profiling commands

The following commands allow you to start, stop, and reset the profiler, and to write profiling data to a file:

`pause` Prompts you to press a key to continue.

`profclear` Resets profiling counts.

`profon` Starts collecting profiling data.

`profoff` Stops collecting profiling data.

`profwrite` Writes profiling information to a file.

### 9.1.8 Miscellaneous commands

These are general commands:

`!` Passes the following command to the host operating system.

`|` Introduces a comment line.

`alias` Defines, undefines, or lists aliases. It enables you to define your own symbolic debugger commands.

`comment` Writes a message to `stderr`.

`help` Displays a list of available commands, or help on a particular command.

`log` Sends the output of subsequent commands to a file as well as the screen.

`obey` Executes a set of debugger commands which have previously been stored in a file, as if they were being typed at the keyboard.

`print` Examines the contents of variables in the program being debugged.

`type` Types the contents of a source file, or any text file, between a specified pair of line numbers.

`while` Is part of a multi-statement line.

`quit` Terminates the current symbolic debugger session and closes any open log or obey files.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.1.9 Commands to access the debug communications channel

`ccin` Selects a file to read.

`ccout` Selects a file to write.

For details of these commands see *Accessing the debug communications channel* on page 9-46.

### 9.1.10 Commands for EmbeddedICE

The following commands support EmbeddedICE. These are deprecated and will not be supported in future versions of the toolkit.

`listconfig` Lists configurations known to the debug agent.

`loadagent` Downloads an EmbeddedICE ROM image.

`loadconfig` Loads an EmbeddedICE configuration data file.

`selectconfig` Selects an EmbeddedICE configuration block.

For details of these commands see *armsd commands for EmbeddedICE* on page 9-47.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## 9.2 Alphabetical list of armsd commands

This section explains how the armsd command syntax is annotated, and lists the terminology used. Every armsd command is then listed and explained, starting with the *! command* on page 9-9.

### 9.2.1 Names used in syntax descriptions

These terms are used in the following sections for the command syntax descriptions:

**Context** The activation state of the program. See *Variable names and context* on page 8-2.

**Expression** An arbitrary expression using the constants, variables, and operators described in *Expressions* on page 8-5. It is either a low-level or a high-level expression, depending on the command.

**Low-level** Low-level expressions are arbitrary expressions using constants, low-level symbols, and operators. You can include high-level variables in low-level expressions if their specification starts with `#` or `$`, or if they are preceded by `^`.

**High-level** High-level expressions are arbitrary expressions using constants, variables, and operators. You can include low-level symbols in high-level expressions by preceding them with `@`.

The `list`, `find`, `examine`, `putfile`, and `getfile` commands require low-level expressions as arguments. All other commands require high-level expressions.

**Location** A location within the program (see *Program locations* on page 8-4).

**Variable** A reference to a variable in the program. Use the simple variable name to look at a variable in the current context, or add more information as described in *Variable names and context* on page 8-2 to see a variable elsewhere in the program.

**Format** This is one of:

- hex.
- ascii.
- string.

  This is a sequence of characters enclosed in double quotes ("). A backslash (\) can be used as an escape character within a string.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

- A C `printf()` function format descriptor. Table 9-1 shows some common descriptors.

**Table 9-1 Format descriptors**

| Type | Format | Description |
|---|---|---|
| int | | Use this only if the expression being printed yields an integer: |
| | `%d` | signed decimal integer (default for integers) |
| | `%u` | unsigned integer |
| | `%x` | hexadecimal (lowercase letters) (same as `hex` format). |
| char | | Use this only if the expression being printed yields an integer: |
| | `%c` | character (same as `ascii` format). |
| char * | | Use this only for expressions which yield a pointer to a zero-terminated string: |
| | `%s` | pointer to character (same as `string` format). |
| void * | | Use this with any kind of pointer: |
| | `%p` | pointer (same as %.8x), for example, 00018abc. |
| float | | Use this only for floating-point results: |
| | `%e` | exponent notation, for example, 9.999999e+00 |
| | `%f` | fixed point notation, for example, 9.999999 |
| | `%g` | general floating-point notation, for example, 1.1, 1.2e+06. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.2 ! command

The `!` command gives access to the command line of the host system without quitting the debugger.

#### Syntax

The syntax of `!` is:

`!`*`command`*

where:

*`command`* Is the operating system command to execute.

#### Usage

Any command whose first character is `!` is passed to the host operating system for execution. For example, `!dir` (DOS) or `!ls` (UNIX) lists the contents of the current directory.

### 9.2.3 | command

The `|` command introduces a comment line.

#### Syntax

The syntax of `|` is:

`|`*`comment`*

where:

*`comment`* Is a text string.

#### Usage

This command enables you to annotate your armsd script file.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.4 alias

The `alias` command defines, undefines, or lists aliases. It enables you to define symbolic debugger commands.

#### Syntax

The syntax of `alias` is:

`alias {name{expansion}}`

where:

*name* Is the name of the alias.

*expansion* Is the expansion for the alias.

#### Usage

If you supply no argument, all currently defined aliases are displayed. If expansion is not specified, the alias named is deleted. Otherwise expansion is assigned to the alias name.

The expansion can be enclosed in double quotes (") to allow the inclusion of characters not normally permitted or with special meanings, such as the alias expansion character (‘) and the statement separator (;).

Aliases are expanded whenever a command line or the command list in a `do` clause is about to be executed.

Words consisting of alphanumeric characters enclosed in backquotes (‘) are expanded. If no corresponding alias is found they are replaced by null strings. If the character following the closing backquote is non-alphanumeric, the closing backquote can be omitted. If the word is the first word of a command, the opening backquote can be omitted. To use a backquote in a command, precede it with another backquote.

#### Example

```
alias restart “reload;break @main;go”
```

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 9.2.5 arguments

The `arguments` command shows the arguments that were passed to the current, or other active procedure.

#### Syntax

The syntax of `arguments` is:

`arguments {context}`

where:

*`context`* Specifies the program context to display. If *`context`* is not specified, the current context is used (normally the procedure active when the program was suspended).

#### Usage

You use the `arguments` command to display the name and context of each argument within the specified context.

### 9.2.6 backtrace

The `backtrace` command prints information about all currently active procedures, starting with the most recent, or for a given number of levels.

#### Syntax

The syntax of `backtrace` is:

`backtrace {count}`

where:

*`count`* Specifies the number of levels to trace. This is an optional argument. If you do not specify *`count`*, the currently active procedures are traced.

#### Usage

When your program has stopped running, because of a breakpoint or watchpoint, you use `backtrace` to extract information on currently active procedures. You can access information like the current function, the line of source code calling the function and so on.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.7 break

The `break` command enables you to specify breakpoints.

#### Syntax

The syntax of the `break` command is:

```
break{/size} {loc {count} {do '{'command{;command}'}'} {if expr}}
```

where:

*/size*
: Specifies which code type to break:
  - `/16` Specifies the instruction size as Thumb.
  - `/32` Specifies the instruction size as ARM.

  If you do not specify *size*, `break` determines the breakpoint size by extracting information from the nearest symbol at or below the address to be broken. This is usually correct, if debug information is available. You must specify *size* when, for example, you set a breakpoint on ROM.

*loc*
: Specifies where to break the code. See *Program locations* on page 8-4.

*count*
: Specifies the number of times the statement must be executed before the program is suspended. It defaults to 1, so if *count* is not specified, the program will be suspended the first time the breakpoint is encountered.

`do`
: Specifies commands to be executed when the breakpoint is reached. Note that these commands must be enclosed in braces, represented above by braces within quotes. Commands must be separated by semicolons. If you not specify a `do` clause, `break` displays the program and source line at the breakpoint. If you want the source line displayed in conjunction with the `do` clause, use `where` as the first command in the `do` clause.

*expr*
: Makes the breakpoint conditional upon the value of *expr*.

#### Usage

The `break` command specifies breakpoints at:

- procedure names
- lines
- statements within a line.

Each breakpoint is given a number prefixed by `#`. A list of current breakpoints and their numbers is displayed if `break` is used without any arguments.

**Note**

Use `unbreak` to delete any unwanted breakpoints. See *unbreak* on page 9-42.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.8 call

The `call` command calls a procedure.

#### Syntax

The syntax of the `call` command is:

`call {/size} loc {(expression-list)}`

where:

*/size*
Specifies whether the procedure is entered in ARM state or Thumb state:

- `/16` specifies Thumb code
- `/32` specifies ARM code.

With no *size* specifier, `call` tries to determine the instruction set of the destination code by extracting information from the nearest symbol at or below the address to call. This usually chooses the correct size, but you can set the size explicitly. The command correctly sets the PSR T-bit to switch to ARM or Thumb state before the call, and restores it on exit.

*loc*
Is a function or low-level address.

*expression_list*
Is a list of arguments to the procedure. String literals are not permitted as arguments. If you specify more than one expression, separate the expressions with commas.

#### Usage

If the procedure (or function) returns a value, examine it using:

`print $result` For integer variables.

`print $fpresult` For floating-point variables.

### 9.2.9 coproc

The `coproc` command describes the register set of a coprocessor and specifies how the contents of the registers are formatted for display.

#### Syntax

The syntax of the `coproc` command is:

`coproc cpnum {rno{:rno1} size access values {displaydesc}*}*`


Copyright © 1999-2001 ARM Limited. All rights reserved.

where:

*cpnum* Identifies the coprocessor.

*rno*{:*rno1*} Identifies the register set.

*size* Is the register size in bytes.

*access* Can comprise the letters:

- R The register is readable.
- W The register is writable.
- D The register is accessed through LDC or STC instructions (if not present, the register is accessed through MRC or MCR instructions).

*values* The format depends on whether the register is to be accessed through MRC/MCR instructions or through LDC/STC instructions. If access is through MRC/MCR instructions, it comprises four integer values separated by a space or comma. These values form bits 0 to 7 and 16 to 23 of an MRC instruction to read the register, and bits 0 to 7 and 16 to 23 of an MCR instruction to write the register:

```
r0_7, r16_23, w0_7, w16_23
```

If access is through LDC/STC instructions, it comprises two integer values to form bits 12 to 15 and bit 22 of LDC or STC instructions to read and write the register:

```
b22, b12_15
```

*displaydesc* Describes how the contents of the registers are to be formatted for display, and takes one of the forms listed in Table 9-2 on page 9-15.

### Usage

Each command can describe one register, or a range of registers, that are accessed and formatted uniformly.

### Example

For example, the floating-point coprocessor might be described by the command:

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

```
coproc 1 0:7 16 RWD 1,8
8 4 RW 0x10,0x30,0x10,0x20 w0[16:20] 'izoux' "_" w0[0:4] 'izoux'
9 4 RW 0x10,0x50,0x10,0x40
```

**Table 9-2 Values for displaydesc argument**

| Item | Definition | | |
|---|---|---|---|
| *string* | Printed as is. | | |
| *field string* | *string* | Used as a `printf` format string to display the value of *field*. | |
| | *field* | One of the forms: | |
| | | *wn* | The whole of the *n*th word of the register value. |
| | | *wn*[*bit*] | Bit *bit* of the *n*th word of the register value. |
| | | *wn*[*bit1*:*bit2*] | Bits *bit1* to *bit2* inclusive of the *n*th word of the register value. You can specify the bits in either order. |
| *field '{' string {string}* '}'* | *field* | One of the forms *wn*[*bit*] or *wn*[*bit1*:*bit2*].<br>There must be one string for each possible value of `field`.<br>The string in the appropriate position for the value of `field` is displayed (the first string for value 0, and so on). | |
| *field 'letters'* | *field* | One of the forms *wn*[*bit*] or *wn*[*bit1*:*bit2*] above.<br>There must be one character in *letters* for each bit of `field`.<br>The letters are displayed in uppercase if the corresponding bit of the field is set, and in lowercase if it is clear.<br>The first letter represents the lowest bit if *bit1* < *bit2*, otherwise it represents the highest bit. | |


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.10 context

The `context` command sets the context in which the variable lookup occurs.

#### Syntax

The syntax of the `context` command is:

`context` *`context`*

where:

*`context`* Specifies the program context. If *`context`* is not specified, the context is reset to the active procedure.

#### Usage

The `context` command affects the default context used by commands that take a context as an argument. When program execution is suspended, the search context is set to the active procedure.

### 9.2.11 cregisters

The `cregisters` command displays the contents of all readable registers of a coprocessor.

#### Syntax

The syntax of the `cregisters` command is:

`cregisters` *`cpnum`*

where

*`cpnum`* Selects the coprocessor.

#### Usage

The contents of the registers is displayed in the format specified by an earlier `coproc` command. The formatting options are described in Table 9-2 on page 9-15.

### 9.2.12 cregdef

The `cregdef` command describes how the contents of a coprocessor register are formatted for display.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Syntax

The syntax of the `cregdef` command is:

`cregdef` *`cpnum rno displaydesc`*

where:

| | |
|---|---|
| *`cpnum`* | Selects the coprocessor. |
| *`rno`* | Selects the register number in the selected coprocessor. |
| *`displaydesc`* | Describes how the processor contents are formatted for display. |

### Usage

The contents of the registers is displayed according to the formatting options described in Table 9-2 on page 9-15.

## 9.2.13 cwrite

The `cwrite` command writes to a coprocessor register.

### Syntax

The syntax of the `cwrite` command is:

`cwrite` *`cpnum rno val`*`{`*`val...`*`}*`

where:

| | |
|---|---|
| *`cpnum`* | Selects the coprocessor. |
| *`rno`* | Selects the register number in the named coprocessor. |
| *`val`* | Each *`val`* is an integer value and there must be one *`val`* item for each word of the coprocessor register. |

### Usage

Before you write to a coprocessor register, you must define that register as writable. This is described in *coproc* on page 9-13.

## 9.2.14 examine

The `examine` command enables you to examine the contents of memory.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Syntax

The syntax of the `examine` command is:

`examine {expression1} {, {+}expression2 }`

where:

*expression1* Gives the start address. The default address used is either:

- the address associated with the current context, minus 64, if the context has changed since the last `examine` command was issued
- the address following the last address displayed by the last `examine` command, if the context has not changed since the last `examine` command was issued.

*expression2* Specifies the end address, which can take three forms:

- if omitted, the end address is the value of the start address +128
- if *expression2* is preceded by +, the end address is given by the value of the start line + *expression2*
- if there is no +, the end line is the value of *expression2.*

You can use the `$examine_lines` variable to alter the default number of lines displayed from its initial value of 8 (128 bytes).

### Usage

This command enables you to examine the contents of the memory between a pair of addresses, displaying it in both hexadecimal and ASCII formats, with 16 bytes per line. Low-level symbols are accepted by default.

## 9.2.15 find

The `find` command finds all occurrences in a specified area of memory of a given integer value or character string.

### Syntax

The syntax of the `find` command is either of the following:

`find expression1,expression2,expression3`

`find string,expression2,expression3`

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

where:

*expression1* Gives the words in memory to search for.

*expression2* Specifies the lower boundary for the search.

*expression3* Specifies the upper boundary for the search.

*string* Specifies the string to search for.

### Usage

If the first form is used, the search is for words in memory whose contents match the value of *expression1*.

If the second form is used, the search is for a sequence of bytes in memory (starting at any byte boundary) whose contents match those of *string*.

Low-level symbols are accepted by default.

## 9.2.16 fpregisters

The `fpregisters` command displays the contents of the eight FPA floating-point registers f0 to f7 and the *Floating Point Status Register* (FPSR).

### Syntax

The syntax of the `fpregisters` command is:

`fpregisters[/full]`

where:

`/full` Includes more information on the floating-point numbers in the registers.

### Usage

There are two formats for the display of floating-point registers.

`fpregisters` Displays the registers and FPSR, in the following form:

```
f0 = 0              f1 = 3.1415926535
f2 = Inf            f3 = 0
f4 = 3.1415926535   f5 = 1
f6 = 0              f7 = 0
fpsr = %IZOux_izoux
```

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

`fpregisters/full`

Produces a more detailed display:

```
f0 = I + 0x3FFF 1 0x0000000000000000
f1 = I + 0x4000 1 0x490FDAA208BA2000
f2 = I +u0x43FF 1 0x0000000000000000
f3 = I - 0x0000 0 0x0000000000000000
f4 = I + 0x4000 1 0x490FDAA208BA2000
f5 = I + 0x3FFF 1 0x0000000000000000
f6 = I + 0x0000 0 0x0000000000000000
f7 = I + 0x0000 1 0x0000000000000000
fpsr = 0x01070000
```

(`fpregisters/full` does not output both sets of values.)

The format of this display is (for example):

*F S Exp* *J Mantissa*`I +u0x43FF 1 0x0000000000000000`

where:

*F* Specifies precision and format:

- `F` Single precision
- `D` Double precision
- `E` Extended precision
- `I` Internal format
- `P` Packed decimal.

*S* Is the sign.

*Exp* Is the exponent.

*J* Is the bit to the left of the binary point.

*Mantissa* Are the digits to the right of the binary point.

*u* The `u` between the sign and the exponent indicates that the number is flagged as *uncommon*, in this example infinity. This applies only to internal format numbers.

In the FPSR description, the first set of letters represent the current settings of the five Exception Trap Enables, also called the Exception Mask. The second set of letters are the Cumulative Exception Flags and represent the exceptions that have occurred. The status of the mask and flag bits is indicated by their case. Uppercase means the flag is set and lowercase means it is cleared.

The exceptions represented are:

- `I` Invalid operation
- `Z` Divide by zero
- `O` Overflow

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

| | |
|---|---|
| `U` | Underflow |
| `X` | Inexact. |

Bits 16 to 20 of the 32-bit FPSR are the Exception Trap Enables, and bits 0 to 4 are the Cumulative Exception Flags.

### 9.2.17 go

The `go` command starts execution of the program.

#### Syntax

The syntax of the `go` command is:

`go {while expression}`

where:

| | |
|---|---|
| `while` | If `while` is used, *`expression`* is evaluated when a breakpoint is reached. If *`expression`* evaluates to true (that is, nonzero), the breakpoint is not reported and execution continues. |
| *`expression`* | Specifies the expression to evaluate. |

#### Usage

The first time `go` is executed, the program starts from its normal entry point. Subsequent `go` commands resume execution from the point at which it was suspended.

### 9.2.18 getfile

The `getfile` command reads from a file and writes the content to memory.

#### Syntax

The syntax of the `getfile` command is:

`getfile filename expression`

where:

| | |
|---|---|
| *`filename`* | Names the file to read from. |
| *`expression`* | Defines the memory location to write to. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

#### Usage

The contents of the file are read as a sequence of bytes, starting at the address which is the value of *expression*. Low-level symbols are accepted by default.

#### Example

```
getfile image.bin 0x0
```

### 9.2.19 help

The `help` command displays a list of available commands, or help on commands.

#### Syntax

The syntax of the `help` command is:

`help {command}`

where:

*command* Is the name of the command you want help on.

#### Usage

The display includes syntax and a brief description of the purpose of each command. If you need information about all commands, as well as their names, type `help *`.

### 9.2.20 in

The `in` command changes the current context by one activation level.

#### Syntax

The syntax of the `in` command is:

`in`

#### Usage

The `in` command sets the context to that called from the current level. It is an error to issue an `in` command when no further movement in that direction is possible.

### 9.2.21 istep

The `istep` command steps execution through one or more instructions.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Syntax

The syntax of the `istep` command is:

`istep {in} {count|w{hile} expression}istep out`

### Usage

This command is analogous to the `step` command except that it steps through one instruction at a time, rather than one high-level language statement at a time.

The use of the `istep` command is not supported in Jazelle state. Submitting this command generates an error message.

## 9.2.22 language

The `language` command sets the high-level language.

### Syntax

The syntax of the `language` command is:

`language {language}`

where:

*language* Specifies the language to use. Enter one of the following:

- `none`
- `C`
- `F77`
- `PASCAL`
- `ASM.`

### Usage

The symbolic debugger uses any high-level debugging tables generated by a compiler to set the default language to the appropriate one for that compiler, whether it is Pascal, Fortran, or C. If it does not find high-level tables, it sets the default language to none, and modifies the behavior of `where` and `step` so that:

`where` Reports the current program counter and instruction.

`step` Steps by one instruction.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.23 let

The `let` command enables you to change the value of a variable or contents of a memory location.

#### Syntax

The syntax of the `let` command is:

```
{let} {variable | location} = expression{{,} expression}*
```

where:

| | |
|---|---|
| *variable* | Names the variable to change. |
| *location* | Names the memory location to change. |
| *expression* | Contains the expression or expressions. |

#### Usage

You use the `let` command in low-level debugging to change memory. If the left-side expression is a constant or a true expression (and not a variable), it is treated as a word address, and memory at that location (and if necessary the following locations) is changed to the values in the following expression(s).

An equals sign (=) or a colon (:) can separate the variable or location from the expression. If you specify multiple expressions, separate them by commas or spaces.

Variables can only be changed to compatible types of expression. However, the debugger converts integers to floating-point and vice versa, rounding to zero. The value of an array can be changed, but not its address, because array names are constants. If the subscript is omitted, it defaults to zero.

If you specify multiple expressions, each expression is assigned to `variable[`*n*`-1]`, where *n* is the nth expression.

See also *let* on page 8-11 for more information on the `let` command.

##### *Specifying the source directory*

You can use the variable `$sourcedir` to specify alternative search paths for source files for the image currently loaded. This variable defaults to NULL if no alternative directories are specified. You can set the value of `$sourcedir` using the command:

```
{let} $sourcedir = string
```

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

where *string* must be a valid pathname, or pathnames. The string must be enclosed in double quotes. If you are using armsd in a Windows DOS environment you must escape the backslash directory separator with another backslash character.

For example:

```
let $sourcedir="c:\\myhome"
```

Multiple paths must be separated by a semicolon. For example:

```
ARMSD: let $sourcedir = "/home/usr/me/src;/home/usr/me/src2"

ARMSD: p $sourcedir
"/home/usr/me/src;/home/usr/me/src2"

ARMSD: let $sourcedir = "/home/usr 2/her name/proj B files"
```

**Note**

No warning is displayed if you enter an invalid pathname.

#### Command-line arguments

You can specify command-line arguments for the debuggee using the `let` command with the root-level variable `$cmdline`. The syntax is:

```
{let} $cmdline = string
```

The program name is automatically passed as the first argument, so you must not include it in the string. You can examine the setting of `$cmdline` using `print`. Commands that use the program name are:

| | |
|---|---|
| `go` | Starts execution of the program. |
| `getfile` | Reads the contents of an area of memory from a file. |
| `load` | Loads an image for debugging. |
| `putfile` | Writes the contents of an area of memory to a file. |
| `reload` | Reloads the object file specified on the armsd command line, or the last load command. |
| `type` | Types the contents of a source file, or any text file, between a specified pair of line numbers. |

#### Reading and writing bytes and halfwords (shorts)

When you specify a write to memory in armsd, a word value is used. For example:

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

```
let 0x8000 = 0x01
```

makes armsd transfer a word (4 bytes) to memory starting at the address `0x8000`. The bytes at `0x8001`, `0x8002`, and `0x8003` are zeroed.

To write only a single byte, you must indicate that a byte transfer is required. You can do this with:

```
let *(char *)0xaddress = value
```

Similarly, to read from an address use:

```
print *(char *)0xaddress
```

You can also read and write halfwords (shorts) in a similar way:

```
let *(short *)0x8000 = valueprint /%x *(short *)0x8000
```

where `/%x` displays in hex.

#### *Editing long long variables*

If you are changing the value of a long long or unsigned long long variable, your new value might be of such a length that it appears to be invalid. In this case, enter `LL` or `ULL` as appropriate at the end of the new value to force its acceptance.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.24 list

The `list` command displays the contents of the memory between a specified pair of addresses in hexadecimal, ASCII, and instruction format, with four bytes (one instruction) per line.

#### Syntax

The syntax of the `list` command is:

`list{/size} {expression1}{, {+}expression2 }`

where:

`size`
: Distinguishes between ARM and Thumb code:

  `/16` Lists as Thumb code

  `/32` Lists as ARM code.

  With no `size` specifier, `list` tries to determine the instruction set of the destination code by extracting information from the nearest symbol at or below the address to start the listing.

*`expression1`*
: Gives the start address. If unspecified, this defaults to either:

  - the address associated with the current context minus 32, if the context has changed since the last `list` command was issued
  - the address following the last address displayed by the last `list` command, if the context has not changed since the last `list` command was issued.

*`expression2`*
: Gives the end address. It can take three forms:

  - if *`expression2`* is omitted, the end address is the value of the start address + 64
  - if it is preceded by +, the end address is the start line + *`expression2`*
  - if there is no +, the end line is the value of *`expression2`*.

#### Usage

The `$list_lines` variable can alter the default number of lines displayed from its initial value of 16 (64 bytes).

Low-level symbols are accepted by default.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.25 load

The `load` command loads an image for debugging.

#### Syntax

The syntax of the `load` command is:

`load{/profile-option} image-file {arguments}`

where:

*profile-option* Specifies which profiling option to use:

- `/callgraph` Directs the debugger to provide the image being loaded with counts which enable the dynamic call-graph profile to be constructed.
- `/profile` Directs the debugger to prepare the image being loaded for flat profiling.

*image-file* Is the name of the file to be debugged.

*arguments* Are the command-line arguments the program normally takes.

#### Usage

You can also specify *image-file* and any necessary *arguments* on the command line when the debugger is invoked. See *Command-line options* on page 7-3 for more information.

If no arguments are supplied, the arguments used in the most recent load or reload, setting of `$cmdline`, or command-line invocation are used again.

The `load` command clears all breakpoints and watchpoints, and does not set a breakpoint at `main()` by default.

If the image you are loading uses floating point data, the `$target_fpu` debugger internal variable must match the image. See Table 8-2 on page 8-7.

### 9.2.26 localvar

The `localvar` command creates a debugger variable of the specified type in the symbol table maintained by the debugger (so access to the variable requires a $ prefix).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Syntax

The syntax of the `localvar` command is:

`localvar` *`vartype varname`*

where:

| | |
|---|---|
| *`vartype`* | Specifies the type of the variable you are creating |
| *`varname`* | Is the name of the variable you are creating. |

### Usage

Use `localvar` to create a local variable, as in the following example that sets the contents of memory from address `0x8000` to address `0x8FFF` to all zeros:

```
localvar int fred
$fred = 0x8000
*$fred = 0; $fred = $fred + 4; while $fred < 0x9000
```


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.27 log

The `log` command sends the output of subsequent commands to a file and to the screen.

#### Syntax

The syntax of the `log` command is:

`log` *`filename`*

where:

*`filename`* Is the name of the file where the record of activity is being stored.

#### Usage

To stop logging, type `log` with no argument. View the file with `type` or a text editor.

**Note**

The debugger prompt and the debug program input/output is not logged.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.28 lsym

The `lsym` command displays low-level symbols and their values.

#### Syntax

The syntax of the `lsym` command is:

`lsym` *`pattern`*

where:

*`pattern`* Is a symbol name or part of a symbol name.

#### Usage

The wildcard (*) matches any number of characters. You can use it at the start of the pattern, at the end, or both:

`lsym *fred` Displays information about fred, alfred.

`lsym fred*` Displays information about fred, frederick.

`lsym *fred*` Displays information about alfred, alfreda, fred, frederick.

The wildcard ? matches one character:

`lsym ??fred` Matches Alfred.

`lsym Jo?` Matches Joe, Joy, and Jon.

### 9.2.29 obey

The `obey` command executes a set of debugger commands that have previously been stored in a file, as if they were being typed at the keyboard.

#### Syntax

The syntax of the `obey` command is:

`obey` *`command-file`*

where:

*`command-file`* Is the file containing the list of commands for execution.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Usage

You can store frequently-used command sequences in files, and call them using `obey`.

## 9.2.30 out

The `out` command changes the current context by one activation level and sets the context to that of the caller of the current context.

### Syntax

The syntax of the `out` command is:

`out`

### Usage

If you issue an `out` command when no further movement in that direction is possible an error message is generated.

If you want to step though assembly language code you must ensure that you use frame directives in your assembly language code to describe stack usage. See the *ADS Assembler Guide* for more information.

## 9.2.31 pause

The `pause` command prompts you to press a key to continue.

### Syntax

The syntax of the `pause` command is:

`pause` *`prompt-string`*

where:

*`prompt-string`* Is a character string written to `stderr`.

### Usage

Execution continues only after you press a key. If you press ESC while commands are being read from a file, the file is closed before execution continues.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 9.2.32 print

The `print` command examines the contents of the variables in the debugged program, or displays the result of arbitrary calculations involving variables and constants.

#### Syntax

The syntax of the `print` command is:

`print{/format} expression`

where:

*/format*
Selects a display format, as described in Table 9-1 on page 9-8. If no */format* string is entered, integer values default to the format described by the variable `$int_format`. Floating-point values use the default format string `%g`. Pointer values are treated as integers, using a default fixed format `%.8x`, for example, `000100e4`.

*expression*
Enters the expression for evaluation.

#### Usage

See also *print* on page 8-11 for more information on the `print` command.

### 9.2.33 profclear

The `profclear` command clears profiling counts.

#### Syntax

The syntax of the `profclear` command is:

`profclear`

#### Usage

For more information on the ARM profiler, refer to the *ADS Linker and Utilities Guide*.

### 9.2.34 profoff

The `profoff` command stops the collection of profiling data.

#### Syntax

The syntax of the `profoff` command is:


Copyright © 1999-2001 ARM Limited. All rights reserved.

`profoff`

#### Usage

For more information on the ARM profiler, refer to the *ADS Linker and Utilities Guide*.

### 9.2.35 profon

The `profon` command starts the collection of profiling data.

#### Syntax

The syntax of the `profon` command is:

`profon {interval}`

where:

*interval* Is the time between PC-sampling in microseconds.

#### Usage

Lower values have a higher performance overhead, and slow down execution, but higher values are not as accurate.

This defaults to flat profiling unless a profile option was specified when the image was loaded. See also *load* on page 9-28.

For more information, see the *ADS Debug Target Guide*.

### 9.2.36 profwrite

The `profwrite` command writes profiling information to a file.

#### Syntax

The syntax of the `profwrite` command is:

`profwrite {filename}`

where:

*filename* Is the name of the file to contain the profiling data.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Usage

The generated information can be viewed using the `armprof` utility. This is described in the *ADS Linker and Utilities Guide*.

## 9.2.37 putfile

The `putfile` command writes the contents of an area of memory to a file. The data is written as a sequence of bytes.

### Syntax

The syntax of the `putfile` command is:

`putfile` *`filename expression1`*`, {+}`*`expression2`*

where:

| | |
|---|---|
| *`filename`* | Specifies the name of the file to write the data into. |
| *`expression1`* | Specifies the lower boundary of the area of memory to be written. |
| *`expression2`* | Specifies the upper boundary of the area of memory to be written. |

### Usage

The upper boundary of the memory area is defined as follows:

- if *`expression2`* is not preceded by a + character, the upper boundary of the memory area is the value of:

  *`expression2`* `- 1`

- if *`expression2`* ispreceded by a + character, the upper boundary of the memory area is the value of:

  *`expression1`* `+` *`expression2`* `- 1.`

Low-level symbols are accepted by default.

### Example

```
putfile image.bin 0x0,+0x8000
```

## 9.2.38 quit

The `quit` command terminates the current armsd session.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Syntax

The syntax of the `quit` command is:

`quit`

### Usage

This command also closes any open log or obey files.

## 9.2.39 readsyms

The `readsyms` command (like the `-symbols` command-line option) reads debug information from the specified image file but does not load the image.

### Syntax

The syntax of the `readsyms` command is:

`readsyms` *`filename`*

### Usage

This command gathers required debugging information from the specified executable image file but does not load the image into memory. The corresponding code must be made available in another way (for example, through a `getfile`, or by being in ROM).

## 9.2.40 registers

The `registers` command displays the contents of ARM registers 0 to 14, the program counter, and the program status registers.

### Syntax

The syntax of the `registers` command is:

`registers {`*`mode`*`}`

where:

*`mode`*
: Selects the registers to display. For a list of mode names, refer to *Predefined symbols* on page 8-14.

  This option can also take the value `all`, where the contents of all registers of the current mode are displayed, together with all banked registers for other modes with the same address width.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### Usage

If used with no arguments, or if *mode* is the current mode, the contents of all registers of the current mode are displayed. If the *mode* argument is specified, but is not the current mode, the contents of the banked registers for that mode are displayed.

A sample display produced by `registers` might look like this:

**Example 9-1**

```
r0  = 0x00000000  r1  = 0x00000001  r2  = 0x00000002  r3  = 0x00000003
r4  = 0x00000004  r5  = 0x00000005  r6  = 0x00000006  r7  = 0x00000007
r8  = 0x00000008  r9  = 0x00000009  r10 = 0x0000000A  r11 = 0x0000000B
r12 = 0x0000000C  r13 = 0x0000000D  r14 = 0x0000000E
pc  = 0x00000000  cpsr = %nzcvqIFt_SVC  spsr = %nzcvqift_Reserved_00
```

## 9.2.41 reload

The `reload` command reloads the object file specified on the armsd command line, or with the last `load` command.

### Syntax

The syntax of the `reload` command is:

`reload`{`/`*profile-option*} {*arguments*}

where

*profile-option*
: Specifies which profiling option to use:

  `/callgraph`
  : Tells the debugger to provide the image being loaded with counts to enable the dynamic call-graph profile to be constructed.

  `/profile`
  : Directs the debugger to prepare the image being loaded for flat profiling.

*arguments*
: Are the command-line arguments the program normally takes. If no *arguments* are specified, the arguments used in the most recent load or reload setting of `$cmdline` or command-line invocation are used again.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### Usage

Breakpoints (but not watchpoints) remain set after a `reload` command.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.42 step

The `step` command steps execution through one or more program statements.

#### Syntax

The syntax of the `step` command is:

`step {in} {out} {count|w{hile} expression}`

where:

| | |
|---|---|
| `in` | Continues single-stepping into procedure calls, so that each statement within a called procedure is single-stepped. If `in` is absent, each procedure call counts as a single statement and is executed without single stepping. |
| `out` | Steps out of a function to the line of originating code that immediately follows that function. |
| *`count`* | Specifies the number of statements to be stepped through. If you omit it only one statement is executed. |
| `while` | Continues single-stepped execution until its *`expression`* evaluates as false (zero). |
| *`expression`* | Is evaluated after every step. |

#### Usage

To step by instructions rather than statements:

- use the `istep` command
- or enter `language none`.

If you want to step though assembly language code you must ensure that you use frame directives in your assembly language code to describe stack usage. See the *ADS Assembler Guide* for more information.

The use of the `step` command is not supported in Jazelle state. Submitting this command generates an error message.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.43 symbols

The `symbols` command lists all symbols defined in the given or current context, with their type information.

#### Syntax

The syntax of the `symbols` command is:

`symbols {context}`

where:

*context* Defines the program context:

- to see global variables, define *context* as the filename with no path or extension
- to see internal variables, use `symbols $`.

#### Usage

The information produced is listed in the form:

*name type*[, *storage-class*], *location*

*storage-class* applies to sourceobject only, not to debugger internal variables, and is one of `auto`, `static`, or `external`.

*location* is one of the following:

- `register r%d` (variable stored in register r%d)
- `memory 0x%x` (variable stored at memory location 0x%x)
- `constant` (variable is actually a constant)
- `debugger variable`
- `filtered`
- `split location` (variable stored in several locations, possibly complex)
- `moving, location` (variable moves, actual location shown)
- `unknown` (location does not exist or an error occurred).

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

### 9.2.44 type

The `type` command types the contents of a source file, or any text file, between a specified pair of line numbers.

#### Syntax

The syntax of the `type` command is:

`type {expression1} {, {{+}expression2} {,filename} }`

where:

*expression1*
Gives the start line. If *expression1* is omitted, it defaults to:
- the source line associated with the current context minus 5, if the context has changed since the last `type` command was issued
- the line following the last line displayed with the `type` command, if the context has not changed.

*expression2*
Gives the end line, in one of three ways:
- if *expression2* is omitted, the end line is the start line +10
- if *expression2* is preceded by `+`, the end line is given by the value of the start line `+` *expression2*
- if there is no `+`, the end line is simply the value of *expression2*.

#### Usage

To look at a file other than that of the current context, specify the filename required and the locations within it.

To change the number of lines displayed from the default setting of 10, use the `$type_lines` variable.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.45 unbreak

The `unbreak` command removes a breakpoint.

#### Syntax

The syntax of the `unbreak` command is:

`unbreak {location | #breakpoint_num}`

where:

| | |
|---|---|
| *location* | Is a source code location. |
| *breakpoint_num* | Is the number of the breakpoint |

#### Usage

If there is only one breakpoint, delete it using `unbreak` without any arguments.

**Note**

A breakpoint always keeps its assigned number. Breakpoints are not renumbered when another breakpoint is deleted, unless the deleted breakpoint was the last one set.

### 9.2.46 unwatch

The `unwatch` command clears a watchpoint.

`unwatch`

#### Syntax

The syntax of the `unwatch` command is:

`unwatch {variable |#watchpoint_number}`

where:

| | |
|---|---|
| *variable* | Is a variable name. |
| *variable* | Is the number of a watchpoint (preceded by #) set using the `watch` command. |

#### Usage

If only one watchpoint has been set, delete it using `unwatch` without any arguments.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.47 variable

The `variable` command provides type and context information on the specified variable (or structure field).

#### Syntax

The syntax of the `variable` command is:

`variable` *`variable`*

where:

*`variable`* Specifies the variable to examine.

#### Usage

The *`variable`* command can also return the type of an expression.

Information about the specified variable is displayed as described in *symbols* on page 9-40.

### 9.2.48 watch

The `watch` command sets a watchpoint on a variable.

#### Syntax

The syntax of the `watch` command is:

`watch {`*`variable`*`}`

where:

*`variable`* Names the variable to watch.

#### Usage

If you do not specify *`variable`*, a list of current watchpoints is displayed along with their numbers. When the variable is altered, program execution is suspended. As with `break` and `unbreak`, these numbers can subsequently be used to remove watchpoints.

Bitfields are not watchable.

If you are debugging through JTAG or EmbeddedICE logic, ensure that watchpoints on global or static variables use hardware watchpoints to avoid any performance penalty.


Copyright © 1999-2001 ARM Limited. All rights reserved.

It is possible to set a watchpoint on a range of addresses. For example:

```
watch (char[16])*0xF200
```

traps all data changes that take place in the 16 bytes of memory starting at `0xF200`.

For this to work efficiently when you are debugging with, for example, Multi-ICE, ensure that the size of the watchpoint in bytes is a power of 2, and that the address of the watchpoint is aligned on a size-byte boundary. Accesses to the area you specify are trapped only if they change any value stored there. A replacement of a value with the same value, for example, is not trapped.

**Note**

Adding software watchpoints can make programs execute very slowly, because the value of variables has to be checked every time they might have been altered. It is more practical to set a breakpoint in the area of suspicion and set watchpoints when execution has stopped.

### 9.2.49 where

The `where` command prints the current context and shows the procedure name, line number in the file, filename, and the line of code.

#### Syntax

The syntax of the `where` command is:

```
where {context}
```

where:

*`context`* Specifies the program context to examine.

#### Usage

If a context is specified after the `where` command, the debugger displays the location of that context.


Copyright © 1999-2001 ARM Limited. All rights reserved.

### 9.2.50 while

The `while` command is only useful at the end of a line containing one or more existing statements. Enter multi-statement lines by separating the statements with ; characters.

#### Syntax

The syntax of the `while` command is:

`statement; {statement;} while` *`expression`*

where:

`statement; {statement;}`

Represents one or more statements to be executed while the expression is true.

*`expression`*

Defines the expression to be evaluated.

#### Usage

After execution of the statements, *`expression`* is evaluated. If true, execution of the line is repeated. This continues until *`expression`* evaluates to false (zero).


Copyright © 1999-2001 ARM Limited. All rights reserved.

## 9.3 Accessing the debug communications channel

The debugger accesses the debug communications channel using the commands described in this section.

For more information, see the *ADS Developer Guide*.

### 9.3.1 ccin

The `ccin` command selects a file containing data for reading into the target.

#### Syntax

The syntax of the `ccin` command is:

`ccin` *`filename`*

where:

*`filename`* Names the file containing the data for reading.

### 9.3.2 ccout

The `ccout` command selects a file where data from the target is written.

#### Syntax

The syntax of the `ccout` command is:

`ccout` *`filename`*

where:

*`filename`* Names the file where the data is written.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## 9.4 armsd commands for EmbeddedICE

The armsd commands described in this section are included for compatibility with EmbeddedICE. These are deprecated, and might be removed from future tool kits.

### 9.4.1 listconfig

The `listconfig` command lists the configurations known to the debug agent.

#### Syntax

The syntax of the `listconfig` command is:

`listconfig` *`file`*

where:

*`file`* Specifies the file where the list of configurations is written.

### 9.4.2 loadagent

The `loadagent` command downloads a replacement EmbeddedICE ROM image, and starts it (in RAM).

#### Syntax

The syntax of the `loadagent` command is:

`loadagent` *`file`*

where:

*`file`* Names the EmbeddedICE ROM image file to load.

### 9.4.3 loadconfig

The `loadconfig` command loads an EmbeddedICE configuration data file. Such files contain data required by EmbeddedICE related to various versions of various processors. See also *selectconfig* on page 9-48.

#### Syntax

The syntax of the `loadconfig` command is:

`loadconfig` *`file`*

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

where:

*file* Names the EmbeddedICE configuration data file to load.

### 9.4.4 selectconfig

An EmbeddedICE configuration data file contains data blocks, each identified by a processor name and version. The `selectconfig` command selects the required block of EmbeddedICE configuration data from those available in the specified configuration file (see *loadconfig* on page 9-47).

#### Syntax

The syntax of the `selectconfig` command is:

`selectconfig` *`name version`*

where:

*name* Is the name of the processor for which configuration data is required.

*version* Indicates the version to be used:

- `any` Accepts any version number. This is the default.
- *n* Uses version *n*.
- *n+* Uses version *n* or later.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

# Appendix A
# AXD and armsd Commands

This appendix compares the commands supported by the command-line interface of AXD with those supported by armsd. It also lists variables with values that you might want to examine or change, showing the AXD commands that enable you to do so. This appendix contains the following sections:


- *Comparison of commands* on page A-2
- *Useful internal variables* on page A-8.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## A.1 Comparison of commands

The ARM debugger armsd is driven by commands only. The ARM debugger AXD is generally driven through its graphical user interface, but it also offers a command-line interface window.

See:

- Chapter 6 *AXD Command-line Interface* for a full description of the commands available in the AXD debugger
- Chapter 9 *Working with armsd* for a full description of the commands available in armsd.

Some commands operate in exactly the same way in both debuggers. Others have close equivalents. Some commands are available in one debugger and not the other. Table A-1 contains all the commands available in both debuggers, arranged alphabetically, and shows equivalences where they exist.

**Table A-1 armsd and AXD commands**

| armsd commands | AXD commands | Short form |
|---|---|---|
| `!command` | - | - |
| `\|comment` | `comment string` | `com` |
| `alias [name[expansion]]` | - | - |
| `arguments [context]` | - | - |
| `backtrace [count]` | `backtrace[ count]`<br>`backtrace` is an alias of `stackentries` | `stk` |
| `break[/size][ loc[ count]`<br>`[ do {command[; command]}]`<br>`[ if expr]]` | `break[ expr\|position [ nth_time]]` | `br` |
| `call[/size] loc[ expr-list]` | - | - |
| - | `cclasses class` | `ccl` |
| - | `cfunctions class` | `cfu` |
| - | `classes[ image]` | `cl` |
| - | `clear` | `clr` |

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

**Table A-1 armsd and AXD commands (continued)**

| armsd commands | AXD commands | Short form |
|---|---|---|
| - | clearbreak *breakpoint*\|all<br>clearbreak has the alias unbreak | cbr |
| - | clearstat *referencepoint* | cstat |
| - | clearwatch *watchpoint*\|all<br>clearwatch has the alias unwatch | cwpt |
| coproc *cpnum*[ *rno*[:*rno1*] *size access values* [ *displaydesc*]*]* | - | - |
| context *context* | context[ *context*] | con |
| - | convariables[ *context*]<br>[ *scope*][ *format*] | convar |
| cregisters *cpnum* | registers *"cpnum"* | reg |
| cregdef *cpnum rno displaydesc* | - | - |
| - | cvariables *class* | cva |
| cwrite *cpnum rno val* [ *val* ...]* | - | - |
| - | dbginternals | di |
| - | disassemble *expr1* [+]*expr2*[ *asm*]<br>disassemble has the alias list | dis |
| let $echo 0\|1 | echo *toggle* | - |
| examine[ *expr1*] [, [+]*expr2*] | examine *expr1*, [+]*expr2*<br>[ *memory*[ *format*]]<br>examine is an alias of memory | mem |
| - | files[ *image*] | fi |
| - | fillmem *expr1* [+]*expr2 value*[ *memory*] | fmem |
| find *expr1*, *expr2*, *expr3* | findvalue *valexpr*[ [*expr1*] [ *expr2*]] | fdv |
| find *string*, *expr2*, *expr3* | findstring *string*[ [*expr1*] [ *expr2*]] | fds |
| - | format[ *fmt_name*[ *ctrl_string*]] | fmt |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table A-1 armsd and AXD commands (continued)**

| armsd commands | AXD commands | Short form |
|---|---|---|
| fpregisters[/*full*] | - | - |
| - | functions[ *image*] | fu |
| getfile *filename expression* | getfile *file addrexpr*<br>getfile is an alias of loadbinary | lb |
| go[ while *expression*] | go[ *processor*]<br>go is an alias of run | r |
| help[ *command*] | help | hlp |
| - | images | im |
| - | imgproperties[ *image*] | ip |
| - | importformat sdm_file[ *fail_action*] | - |
| in | stackin | in |
| istep[ in][ count\|w[hile] *expr*] istep out | - | - |
| language[ *language*] | - | - |
| [let] [*variable*\|*location*] = *expression* [[,] *expression*]* | let *expr1*, *expr2*<br>let is an alias of setwatch | swat |
| list[/*size*][ *expr1*] [, [+]*expr2*] | list *expr1* [+]*expr2*[ *asm*]<br>list is an alias of disassemble | dis |
| - | listformat[ *nbits*] | lsfmt |
| load[/*profile-opt*] *image-file*[ *args*] | load *file*[ *processor*] | ld |
| - | loadbinary *file addrexpr*<br>loadbinary has the alias getfile | lb |
| - | loadsession *sesfile* | lss |
| - | loadsymbols *file*[ *processor*]<br>loadsymbols has the alias readsyms | lds |
| localvar *vartype varname* | - | - |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table A-1 armsd and AXD commands (continued)**

| armsd commands | AXD commands | Short form |
|---|---|---|
| log *filename* | log[ *file*] | - |
| lsym *pattern* | lowlevel[ *image*] | lsym |
| - | memory *expr1* [+]*expr2* [ *memory*[ *format*]]<br>memory has the alias examine | mem |
| obey *command-file* | obey *file* | - |
| out | stackout | out |
| - | parse *toggle* | par |
| pause *prompt-string* | - | - |
| print[/*format*] *expression* | print *expr*[ *format*]<br>print is an alias of watch | wat |
| - | processors | proc |
| - | procproperties[ *image*] | pp |
| profclear | - | - |
| profoff | - | - |
| profon[ *interval*] | - | - |
| profwrite[ *filename*] | - | - |
| putfile *filename expr1*, [+]*expr2* | putfile *file expr1* [+]*expr2*<br>putfile is an alias of savebinary | sb |
| quit | quitdebugger | quitd |
| readsyms *filename* | readsyms *file*[ *processor*]<br>readsyms is an alias of loadsymbols | lds |
| - | record[ *file*] | rec |
| - | regbanks[ *processor*] | regbk |
| - | registers[ *regbank*[ *format*]] | reg |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table A-1 armsd and AXD commands (continued)**

| armsd commands | AXD commands | Short form |
|---|---|---|
| reload[/*profile-option*] [ *arguments*] | reload[ *image*] | rld |
| - | run[ *processor*]<br>run has the alias go | r |
| - | runtopos *position*[ *processor*] | rto |
| - | savebinary *file expr1* [+]*expr2*<br>savebinary has the alias putfile | sb |
| - | savesession *sesfile* | ss |
| - | setaci *string* | aci |
| - | setbreakprops *breakpoint propid value* | - |
| - | setimgprop *image ipvar value* | sip |
| - | setmem *addrexpr valexpr* [ *memory*] | smem |
| pc=xx | setpc *expr* | pc |
| - | setproc *processor* | sproc |
| - | setprocprop *ppvar value* | spp |
| - | setreg [*regbank*\|]*register expr* | sreg |
| - | setsourcedir *directory_list* | ssd |
| - | setwatch *expr1*, *expr2*<br>setwatch has the alias let | swat |
| - | setwatchprops *watchpoint propid value* | swp |
| - | source *value1* [+]*value2* [ *file*]<br>source has the alias type | src |
| - | sourcedir[ *path*[ *index*]] | sdir |
| backtrace [*count*] | stackentries[ *count*]<br>stackentries has the alias backtrace | stk |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table A-1 armsd and AXD commands (continued)**

| armsd commands | AXD commands | Short form |
|---|---|---|
| in | stackin | in |
| out | stackout | out |
| p $statistics | statistics[ *ref_pt_name*] | stat |
| step [in\|out] [ *count*\|w[hile] *expr*] | step[ *step*][ *instr*] | st |
| - | stepsize[ *instr*] | ssize |
| - | stop[ *processor*] | - |
| symbols[ *context*] | - | - |
| - | trace on\|off | trace |
| - | traceload *tcfile* | trload |
| type[ *expr1*][, [[+]*expr2*] [, *fname*]] | type *value1* [+]*value2*[ *file*]<br>type is an alias of source | src |
| unbreak [ *location*\|*#breakpoint_num*] | unbreak *breakpoint*<br>unbreak is an alias of clearbreak | cbr |
| unwatch [ *variable*\|*#watchpoint_num*] | unwatch *watchpoint*<br>unwatch is an alias of clearwatch | cwpt |
| variable *variable* | variables[ *image*] | va |
| print *expr* | watch *expr*[ *format*]<br>watch has the alias print | wat |
| watch[ *variable*] | watchpt[ *expr*[ *nth_time*]] | wpt |
| where[ *context*] | where[ *context*] | - |
| *statement*;[ *statement*;] while *expr* | - | - |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## A.2 Useful internal variables

Table A-2 lists some variables with values that you might want to examine or change.

In armsd you examine these as debugger internal variables and can change their values with a `let` command. In AXD more CLI commands are available. These are described in full in Chapter 6 *AXD Command-line Interface*.

**Table A-2 Internal variables**

| armsd variable | AXD command |
|---|---|
| `$vector_catch` | `pp` to examine, `spp` to change |
| `$cmdline` | `setimgprop` *`image`* `cmdline` *`params`* |
| `$rdi_log` | `pr` to examine, `let` to change |
| `$target_fpu` | `pr` to examine, `let` to change |
| `$semihosting_enabled` | `pp` to examine, `spp` to change |
| `$semihosting_vector` | `pp` to examine, `spp` to change |
| `$semihosting_arm_swi` | `pp` to examine, `spp` to change |
| `$semihosting_thumb_swi` | `pp` to examine, `spp` to change |
| `$arm_swi` | `setprocprop arm_semihosting_swi` *`value`* |
| `$thumb_swi` | `setprocprop thumb_semihosting_swi` *`value`* |
| `$semihosting_dcchandler_ address` | `pp` to examine, `spp` to change |
| `$icebreaker_lockedpoints` | `pr` to examine, `let` to change |
| `$safe_non_vector_address` | `pr` to examine, `let` to change |
| `$top_of_memory` | `pr` to examine, `let` to change |
| `$system_reset` | `pr` to examine, `let` to change |
| `$cp_access_code_address` | `pr` to examine, `let` to change. Multi-ICE only. |
| `$user_input_bit1` | Hardware input to Multi-ICE only. Not writable. |
| `$user_input_bit2` | Hardware input to Multi-ICE only. Not writable. |
| `$user_output_bit1` | `pr` to examine, `let` to change |
| `$user_outout_bit2` | `pr` to examine, `let` to change |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table A-2 Internal variables (continued)**

| armsd variable | AXD command |
|---|---|
| $arm9_restart_code_address | `pr` to examine, `let` to change. Multi-ICE 1.3 and 1.4 |
| $cache_clean_code_address | `pr` to examine, `let` to change. Multi-ICE 2.0. |
| `$sw_breakpoints_preferred` | `pr` to examine, `let` to change. Multi-ICE only. |
| `$sourcedir` | `sdir` to examine, `ssd` to change |
| `$echo` | `echo on\|off` |


Copyright © 1999-2001 ARM Limited. All rights reserved.

Copyright © 1999-2001 ARM Limited. All rights reserved.

# Appendix B
# Coprocessor Registers

This appendix describes coprocessor registers for various ARM processors. It contains the following sections:


- *ARM710T processor* on page B-2
- *ARM720T processor* on page B-3
- *ARM740T processor* on page B-4
- *ARM920T Rev 0 processor* on page B-5
- *ARM920T Rev 1 processor* on page B-7
- *ARM940T Rev 0 processor* on page B-9
- *ARM940T Rev 1 processor* on page B-11
- *ARM946E-S processor* on page B-13
- *ARM966E-S processor* on page B-15
- *ARM10200E processor* on page B-16
- *ARM1020E processor* on page B-20
- *ARM10E processor* on page B-22
- *XScale processor* on page B-24.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## B.1 ARM710T processor

Table B-1 describes the coprocessor registers of the ARM710T processor.

**Table B-1 ARM710T**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DACR | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate | Invalidate cache | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate | Invalidate TLB | CP = 15: CRn = 8, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_Address | Invalidate TLB single entry (by address) | CP = 15: CRn = 8, CRm = 7, op_1 = 0, op_2 = 1 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.2 ARM720T processor

Table B-2 describes the coprocessor registers of the ARM720T processor.

**Table B-2 ARM720T**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DACR | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate | Invalidate cache | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate | Invalidate TLB | CP = 15: CRn = 8, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_Address | Invalidate TLB single entry (by address) | CP = 15: CRn = 8, CRm = 7, op_1 = 0, op_2 = 1 |
| CP15: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.3 ARM740T processor

Table B-3 describes the coprocessor registers of the ARM740T processor.

**Table B-3 ARM740T**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cacheable | Cacheable | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Bufferable | Bufferable | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Protection | Protection | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region0 | Memory area 0 definition | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region1 | Memory area 1 definition | CP = 15: CRn = 6, CRm = 1, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region2 | Memory area 2 definition | CP = 15: CRn = 6, CRm = 2, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region3 | Memory area 3 definition | CP = 15: CRn = 6, CRm = 3, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region4 | Memory area 4 definition | CP = 15: CRn = 6, CRm = 4, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region5 | Memory area 5 definition | CP = 15: CRn = 6, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region6 | Memory area 6 definition | CP = 15: CRn = 6, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region7 | Memory area 7 definition | CP = 15: CRn = 6, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate | Invalidate cache | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.4 ARM920T Rev 0 processor

Table B-4 describes the coprocessor registers of the ARM920T Rev 0 processor.

**Table B-4 ARM920T Rev 0**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DACR | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DLOCK | Data cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ILOCK | Instruction cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: TLBDLOCK | Data TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TLBILOCK | Instruction TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate | Invalidate both caches | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 1 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-4 ARM920T Rev 0 (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Index | Clean D cache single index | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: CleanInvalidate_D_Index | Clean and invalidate D cache single index | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |
| CP15: TLB operations: Invalidate | Invalidate I+D TLB | CP = 15: CRn = 8, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_I | Invalidate I TLB | CP = 15: CRn = 8, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_I_Address | Invalidate I TLB entry (by address) | CP = 15: CRn = 8, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: TLB operations: Invalidate_D | Invalidate D TLB | CP = 15: CRn = 8, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_D_Address | Invalidate D TLB entry (by address) | CP = 15: CRn = 8, CRm = 6, op_1 = 0, op_2 = 1 |
| CP15: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.5 ARM920T Rev 1 processor

Table B-5 describes the coprocessor registers of the ARM920T Rev 1 processor.

**Table B-5 ARM920T Rev 1**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DACR | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: PFSR | Prefetch fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: FAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DLOCK | Data cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ILOCK | Instruction cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: TLBDLOCK | Data TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TLBILOCK | Instruction TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate | Invalidate both caches | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 1 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-5 ARM920T Rev 1 (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Index | Clean D cache single index | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: CleanInvalidate_D_Index | Clean and invalidate D cache single index | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |
| CP15: TLB operations: Invalidate | Invalidate I+D TLB | CP = 15: CRn = 8, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_I | Invalidate I TLB | CP = 15: CRn = 8, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_I_Address | Invalidate I TLB entry (by address) | CP = 15: CRn = 8, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: TLB operations: Invalidate_D | Invalidate D TLB | CP = 15: CRn = 8, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: TLB operations: Invalidate_D_Address | Invalidate D TLB entry (by address) | CP = 15: CRn = 8, CRm = 6, op_1 = 0, op_2 = 1 |
| CP15: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.6 ARM940T Rev 0 processor

Table B-6 describes the coprocessor registers of the ARM940T Rev 0 processor.

**Table B-6 ARM940T Rev 0**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DCacheable | Data cacheable | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ICacheable | Instruction cacheable | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Bufferable | Bufferable | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DProtection | Data protection | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: IProtection | Instruction protection | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Data Regions: DRegion0 | Data memory area 0 definition | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion1 | Data memory area 1 definition | CP = 15: CRn = 6, CRm = 1, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion2 | Data memory area 2 definition | CP = 15: CRn = 6, CRm = 2, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion3 | Data memory area 3 definition | CP = 15: CRn = 6, CRm = 3, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion4 | Data memory area 4 definition | CP = 15: CRn = 6, CRm = 4, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion5 | Data memory area 5 definition | CP = 15: CRn = 6, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion6 | Data memory area 6 definition | CP = 15: CRn = 6, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion7 | Data memory area 7 definition | CP = 15: CRn = 6, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion0 | Instruction memory area 0 definition | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion1 | Instruction memory area 1 definition | CP = 15: CRn = 6, CRm = 1, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion2 | Instruction memory area 2 definition | CP = 15: CRn = 6, CRm = 2, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion3 | Instruction memory area 3 definition | CP = 15: CRn = 6, CRm = 3, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion4 | Instruction memory area 4 definition | CP = 15: CRn = 6, CRm = 4, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-6 ARM940T Rev 0 (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Instruction Regions: IRegion5 | Instruction memory area 5 definition | CP = 15: CRn = 6, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion6 | Instruction memory area 6 definition | CP = 15: CRn = 6, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion7 | Instruction memory area 7 definition | CP = 15: CRn = 6, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 8, op_1 = 0, op_2 = 2 |
| CP15: Cache lockdown: D_Lockdown | Data lockdown control | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache lockdown: I_Lockdown | Instruction lockdown control | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.7 ARM940T Rev 1 processor

Table B-7 describes the coprocessor registers of the ARM940T Rev 1 processor.

**Table B-7 ARM940T Rev 1**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DCacheable | Data cacheable | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ICacheable | Instruction cacheable | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Bufferable | Bufferable | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DProtection | Data protection | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: IProtection | Instruction protection | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Data Regions: DRegion0 | Data memory area 0 definition | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion1 | Data memory area 1 definition | CP = 15: CRn = 6, CRm = 1, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion2 | Data memory area 2 definition | CP = 15: CRn = 6, CRm = 2, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion3 | Data memory area 3 definition | CP = 15: CRn = 6, CRm = 3, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion4 | Data memory area 4 definition | CP = 15: CRn = 6, CRm = 4, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion5 | Data memory area 5 definition | CP = 15: CRn = 6, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion6 | Data memory area 6 definition | CP = 15: CRn = 6, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Data Regions: DRegion7 | Data memory area 7 definition | CP = 15: CRn = 6, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion0 | Instruction memory area 0 definition | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion1 | Instruction memory area 1 definition | CP = 15: CRn = 6, CRm = 1, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion2 | Instruction memory area 2 definition | CP = 15: CRn = 6, CRm = 2, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion3 | Instruction memory area 3 definition | CP = 15: CRn = 6, CRm = 3, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-7 ARM940T Rev 1 (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Instruction Regions: IRegion4 | Instruction memory area 4 definition | CP = 15: CRn = 6, CRm = 4, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion5 | Instruction memory area 5 definition | CP = 15: CRn = 6, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion6 | Instruction memory area 6 definition | CP = 15: CRn = 6, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Instruction Regions: IRegion7 | Instruction memory area 7 definition | CP = 15: CRn = 6, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 8, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| CP15: Cache lockdown: D_Lockdown | Data lockdown control | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache lockdown: I_Lockdown | Instruction lockdown control | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.8 ARM946E-S processor

Table B-8 describes the coprocessor registers of the ARM946E-S processor.

**Table B-8 ARM946E-S**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: TCMS | Tightly coupled memory size | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 2 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DCacheable | Data cacheable | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ICacheable | Instruction cacheable | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Bufferable | Bufferable | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DProtection | Data protection | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 2 |
| CP15: IProtection | Instruction protection | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 3 |
| CP15: Protection Regions: Region0 | Memory area 0 definition | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region1 | Memory area 1 definition | CP = 15: CRn = 6, CRm = 1, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region2 | Memory area 2 definition | CP = 15: CRn = 6, CRm = 2, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region3 | Memory area 3 definition | CP = 15: CRn = 6, CRm = 3, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region4 | Memory area 4 definition | CP = 15: CRn = 6, CRm = 4, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region5 | Memory area 5 definition | CP = 15: CRn = 6, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region6 | Memory area 6 definition | CP = 15: CRn = 6, CRm = 6, op_1 = 0, op_2 = 0 |
| CP15: Protection Regions: Region7 | Memory area 7 definition | CP = 15: CRn = 6, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-8 ARM946E-S (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Index | Clean D cache single index | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Index | Clean and invalidate D cache single index | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |
| CP15: Cache operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| CP15: Cache lockdown: D_Lockdown | Data lockdown control | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache lockdown: I_Lockdown | Instruction lockdown control | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Tightly coupled regions: DTCMR | Data tightly coupled memory region | CP = 15: CRn = 9, CRm = 1, op_1 = 0, op_2 = 0 |
| CP15: Tightly coupled regions: ITCMR | Instruction tightly coupled memory region | CP = 15: CRn = 9, CRm = 1, op_1 = 0, op_2 = 1 |
| CP15: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.9 ARM966E-S processor

Table B-9 describes the coprocessor registers of the ARM966E-S processor.

**Table B-9 ARM966E-S**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |
| CP15: Operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.10 ARM10200E processor

Table B-10 describes the coprocessor registers of the ARM10200E processor.

**Table B-10 ARM10200E**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DACR | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DFAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: IFAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: DLOCK | Data cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ILOCK | Instruction cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: TLBDLOCK | Data TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TLBILOCK | Instruction TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate | Invalidate both caches | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-10 ARM10200E (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Index | Clean D cache single index | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: CleanInvalidate_D_Index | Clean and invalidate D cache single index | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S0 | | CP = 11: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S1 | | CP = 11: CRn = 0, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S2 | | CP = 11: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S3 | | CP = 11: CRn = 1, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S4 | | CP = 11: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S5 | | CP = 11: CRn = 2, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S6 | | CP = 11: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S7 | | CP = 11: CRn = 3, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S8 | | CP = 11: CRn = 4, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S9 | | CP = 11: CRn = 4, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S10 | | CP = 11: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S11 | | CP = 11: CRn = 5, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S12 | | CP = 11: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S13 | | CP = 11: CRn = 6, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S14 | | CP = 11: CRn = 7, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-10 ARM10200E (continued)**

| Name | Description | Register |
|---|---|---|
| VFP: VFP (Single): S15 | | CP = 11: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S16 | | CP = 11: CRn = 8, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S17 | | CP = 11: CRn = 8, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S18 | | CP = 11: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S19 | | CP = 11: CRn = 9, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S20 | | CP = 11: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S21 | | CP = 11: CRn = 10, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S22 | | CP = 11: CRn = 11, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S23 | | CP = 11: CRn = 11, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S24 | | CP = 11: CRn = 12, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S25 | | CP = 11: CRn = 12, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S26 | | CP = 11: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S27 | | CP = 11: CRn = 13, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S28 | | CP = 11: CRn = 14, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S29 | | CP = 11: CRn = 14, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Single): S30 | | CP = 11: CRn = 15, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Single): S31 | | CP = 11: CRn = 15, CRm = 0, op_1 = 0, op_2 = 4 |
| VFP: VFP (Double): D0 | | CP = 11: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-10 ARM10200E (continued)**

| Name | Description | Register |
|---|---|---|
| VFP: VFP (Double): D1 | | CP = 11: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D2 | | CP = 11: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D3 | | CP = 11: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D4 | | CP = 11: CRn = 4, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D5 | | CP = 11: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D6 | | CP = 11: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D7 | | CP = 11: CRn = 7, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D8 | | CP = 11: CRn = 8, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D9 | | CP = 11: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D10 | | CP = 11: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D11 | | CP = 11: CRn = 11, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D12 | | CP = 11: CRn = 12, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D13 | | CP = 11: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D14 | | CP = 11: CRn = 14, CRm = 0, op_1 = 0, op_2 = 0 |
| VFP: VFP (Double): D15 | | CP = 11: CRn = 15, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.11 ARM1020E processor

Table B-11 describes the coprocessor registers of the ARM1020E processor.

**Table B-11 ARM1020E**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DACR | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DFAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: IFAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: DLOCK | Data cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ILOCK | Instruction cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: TLBDLOCK | Data TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TLBILOCK | Instruction TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate | Invalidate both caches | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-11 ARM1020E (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Index | Clean D cache single index | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: CleanInvalidate_D_Index | Clean and invalidate D cache single index | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.12 ARM10E processor

Table B-12 describes the coprocessor registers of the ARM10E processor.

**Table B-12 ARM10E**

| Name | Description | Register |
|---|---|---|
| CP15: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DACR | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: DFAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: IFAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: DLOCK | Data cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: ILOCK | Instruction cache lockdown | CP = 15: CRn = 9, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: TLBDLOCK | Data TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: TLBILOCK | Instruction TLB lockdown | CP = 15: CRn = 10, CRm = 0, op_1 = 0, op_2 = 1 |
| CP15: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |
| CP15: Cache operations:Invalidate | Invalidate both caches | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I | Invalidate entire I cache | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| CP15: Cache operations: Invalidate_I_Address | Invalidate I cache single entry (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Prefetch_I | Prefetch I cache line | CP = 15: CRn = 7, CRm = 13, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Invalidate_D | Invalidate entire D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-12 ARM10E (continued)**

| Name | Description | Register |
|---|---|---|
| CP15: Cache operations: Invalidate_D_Address | Invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Address | Clean D cache single entry (by address) | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: CleanInvalidate_D_Address | Clean and invalidate D cache single entry (by address) | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 1 |
| CP15: Cache operations: Clean_D_Index | Clean D cache single index | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: CleanInvalidate_D_Index | Clean and invalidate D cache single index | CP = 15: CRn = 7, CRm = 14, op_1 = 0, op_2 = 2 |
| CP15: Cache operations: Drain | Drain write buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| CP15: Cache operations: Wait | Wait for interrupt | CP = 15: CRn = 7, CRm = 0, op_1 = 0, op_2 = 4 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## B.13 XScale processor

Table B-13 describes the coprocessor registers of the XScale processor.

**Table B-13 XScale**

| Name | Description | Register |
|---|---|---|
| Accumulators: ACC0 | Accumulator 0 | CP = 15: CRn = 15, CRm = 1, op_1 = 0, op_2 = 0 |
| Interrupt Controller: INTCTL | Interrupt control register | CP = 15: CRn = 15, CRm = 1, op_1 = 0, op_2 = 0 |
| Interrupt Controller: INTSRC | Interrupt source register | CP = 15: CRn = 15, CRm = 1, op_1 = 0, op_2 = 0 |
| Interrupt Controller: INTSTR | Interrupt steer register | CP = 15: CRn = 15, CRm = 1, op_1 = 0, op_2 = 0 |
| Performance Monitors: PMNC | Performance monitor control register | CP = 14: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| Performance Monitors: CCNT | Clock count register | CP = 14: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| Performance Monitors: PMN0 | Performance count register 0 | CP = 14: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| Performance Monitors: PMN1 | Performance count register 1 | CP = 14: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| Software Debug: IBCR0 | Instruction breakpoint and control register 0 | CP = 15: CRn = 14, CRm = 8, op_1 = 0, op_2 = 0 |
| Software Debug: IBCR1 | Instruction breakpoint and control register 1 | CP = 15: CRn = 14, CRm = 9, op_1 = 0, op_2 = 0 |
| Software Debug: DBR0 | Data breakpoint register 0 | CP = 15: CRn = 14, CRm = 0, op_1 = 0, op_2 = 0 |
| Software Debug: DBR1 | Data breakpoint register 1 | CP = 15: CRn = 14, CRm = 3, op_1 = 0, op_2 = 0 |
| Software Debug: DBCON | Data breakpoint controls register | CP = 15: CRn = 14, CRm = 4, op_1 = 0, op_2 = 0 |
| Software Debug: TX | Transmit register | CP = 14: CRn = 8, CRm = 0, op_1 = 0, op_2 = 0 |
| Software Debug: RX | Receive register | CP = 14: CRn = 9, CRm = 0, op_1 = 0, op_2 = 0 |
| Software Debug: DCSR | Debug control and status register | CP = 14: CRn = 10, CRm = 0, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-13 XScale (continued)**

| Name | Description | Register |
|---|---|---|
| Software Debug: CHKPT0 | Checkpoint register 0 | CP = 14: CRn = 12, CRm = 0, op_1 = 0, op_2 = 0 |
| Software Debug: CHKPT1 | Checkpoint register 1 | CP = 14: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |
| Software Debug: TXRXCTRL | TX RX control register | CP = 14: CRn = 14, CRm = 0, op_1 = 0, op_2 = 0 |
| Clock and Power: CCLKCFG | Core clock configuration register | CP = 14: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| Clock and Power: PWRMODE | Power mode register | CP = 14: CRn = 7, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: ID | Chip ID | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: Cache type | Cache type | CP = 15: CRn = 0, CRm = 0, op_1 = 0, op_2 = 1 |
| System Control: Control | Control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: Aux Control | Auxiliary control | CP = 15: CRn = 1, CRm = 0, op_1 = 0, op_2 = 1 |
| System Control: TTBR | Translation table base register | CP = 15: CRn = 2, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: DAC | Domain access control register | CP = 15: CRn = 3, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: FSR | Fault status register | CP = 15: CRn = 5, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: FAR | Fault address register | CP = 15: CRn = 6, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: PID | Process ID register | CP = 15: CRn = 13, CRm = 0, op_1 = 0, op_2 = 0 |
| System Control: CP_Access | Coprocessor access register | CP = 15: CRn = 15, CRm = 1, op_1 = 0, op_2 = 0 |
| System Control: Cache operations: Invalidate | Invalidate I+D cache and BTB | CP = 15: CRn = 7, CRm = 7, op_1 = 0, op_2 = 0 |
| System Control: Cache operations: Invalidate_I | Invalidate I cache and BTB | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 0 |
| System Control: Cache operations: Invalidate_I_Address | Invalidate I cache line (by address) | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 1 |
| System Control: Cache operations: Invalidate_D | Invalidate D cache | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-13 XScale (continued)**

| Name | Description | Register |
|---|---|---|
| System Control: Cache operations: Invalidate_D_Address | Invalidate D cache line | CP = 15: CRn = 7, CRm = 6, op_1 = 0, op_2 = 1 |
| System Control: Cache operations: Clean_D | Clean D cache line | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 1 |
| System Control: Cache operations: Drain | Drain write (and fill) buffer | CP = 15: CRn = 7, CRm = 10, op_1 = 0, op_2 = 4 |
| System Control: Cache operations: Invalidate_BTB | Invalidate branch target buffer | CP = 15: CRn = 7, CRm = 5, op_1 = 0, op_2 = 6 |
| System Control: Cache operations: Allocate_D_Address | Allocate line in the D cache | CP = 15: CRn = 7, CRm = 2, op_1 = 0, op_2 = 5 |
| System Control: TLB operations: Invalidate | Invalidate I+D TLB | CP = 15: CRn = 8, CRm = 7, op_1 = 0, op_2 = 0 |
| System Control: TLB operations: Invalidate_I | Invalidate I TLB | CP = 15: CRn = 8, CRm = 5, op_1 = 0, op_2 = 0 |
| System Control: TLB operations: Invalidate_I_Address | Invalidate I TLB entry (by address) | CP = 15: CRn = 8, CRm = 5, op_1 = 0, op_2 = 1 |
| System Control: TLB operations: Invalidate_D | Invalidate D TLB | CP = 15: CRn = 8, CRm = 6, op_1 = 0, op_2 = 0 |
| System Control: TLB operations: Invalidate_D_Address | Invalidate D TLB entry (by address) | CP = 15: CRn = 8, CRm = 6, op_1 = 0, op_2 = 1 |
| System Control: Cache lockdown: FetchLock_I | Fetch and lock I cache line | CP = 15: CRn = 9, CRm = 1, op_1 = 0, op_2 = 0 |
| System Control: Cache lockdown: Unlock_I | Unlock I cache | CP = 15: CRn = 9, CRm = 1, op_1 = 0, op_2 = 1 |
| System Control: Cache lockdown: Lock_D | D cache lock register | CP = 15: CRn = 9, CRm = 2, op_1 = 0, op_2 = 0 |
| System Control: Cache lockdown: Unlock_D | Unlock data cache | CP = 15: CRn = 9, CRm = 2, op_1 = 0, op_2 = 1 |
| System Control: TLB lockdown: TranslateLock_I_Address | Translate and lock I TLB entry (by address) | CP = 15: CRn = 10, CRm = 4, op_1 = 0, op_2 = 0 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table B-13 XScale (continued)**

| Name | Description | Register |
|---|---|---|
| System Control: TLB lockdown: Unlock_I | Unlock I TLB | CP = 15: CRn = 10, CRm = 4, op_1 = 0, op_2 = 1 |
| System Control: TLB lockdown: TranslateLock_D_Address | Translate and lock D TLB entry (by address) | CP = 15: CRn = 10, CRm = 8, op_1 = 0, op_2 = 0 |
| System Control: TLB lockdown: Unlock_D | Unlock D TLB | CP = 15: CRn = 10, CRm = 8, op_1 = 0, op_2 = 1 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

Copyright © 1999-2001 ARM Limited. All rights reserved.

# Appendix C
# Supplementary Display Module Formats

This appendix describes the structure and content of *Supplementary Display Module* (SDM) files that contain display format definitions. You can read SDM files in AXD with the `importformat` command. The file `armperip.xml`, described in the *ADS Debug Target Guide*, can also contain these display format definitions. This appendix contains the following sections:


- *Predefined formats* on page C-2
- *User-defined formats* on page C-5.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## C.1 Predefined formats

Some CLI commands in AXD take a display format name as an argument. Table C-1 to Table C-6 on page C-4 list the predefined display format names that are always valid in AXD.

**Table C-1 8-bit data display formats**

| Format description | Format name |
|---|---|
| Hexadecimal | hex_8 |
| Decimal | dec_8 |
| Unsigned decimal | udec_8 |
| Octal | oct_8 |
| Binary | bin_8 |
| ASCII | ascii_8 |
| Hex, no prefix | hex_noprefix_8 |
| Octal, no prefix | oct_noprefix_8 |

**Table C-2 16-bit data display formats**

| Format description | Format name |
|---|---|
| Hexadecimal | hex_16 |
| Decimal | dec_16 |
| Unsigned decimal | udec_16 |
| Octal | oct_16 |
| Binary | bin_16 |
| ASCII | ascii_16 |
| Hex, no prefix | hex_noprefix_16 |
| Octal, no prefix | oct_noprefix_16 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table C-3 32-bit data display formats**

| Format description | Format name |
|---|---|
| Hexadecimal | hex_32 |
| Decimal | dec_32 |
| Single | fp_32 |
| Scientific single | fp_sci_32 |
| PSR | psr |
| JPSR | jpsr |
| EPSR | epsr |
| FPSR | fpsr |
| Unsigned decimal | udec_32 |
| Octal | oct_32 |
| Binary | bin_32 |
| ASCII | ascii_32 |
| Hex, no prefix | hex_noprefix_32 |
| Octal, no prefix | oct_noprefix_32 |

**Table C-4 40-bit data display formats**

| Format description | Format name |
|---|---|
| Hexadecimal | hex_40 |
| Decimal | dec_40 |
| Unsigned decimal | udec_40 |
| Octal | oct_40 |
| Binary | bin_40 |
| ASCII | ascii_40 |
| Hex, no prefix | hex_noprefix_40 |
| Octal, no prefix | oct_noprefix_40 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table C-5 64-bit data display formats**

| Format description | Format name |
|---|---|
| Hexadecimal | hex_64 |
| Decimal | dec_64 |
| Double | fp_64 |
| Scientific double | fp_sci_64 |
| Unsigned decimal | udec_64 |
| Octal | oct_64 |
| Binary | bin_64 |
| ASCII | ascii_64 |
| Hex, no prefix | hex_noprefix_64 |
| Octal, no prefix | oct_noprefix_64 |

**Table C-6 80-bit data display formats**

| Format description | Format name |
|---|---|
| Raw floating point | fp_sci_80 |


Copyright © 1999-2001 ARM Limited. All rights reserved.

## C.2 User-defined formats

Some CLI commands in AXD take a display format name as an argument. As well as predefined formats, you can define your own formats in a `.sdm` (Supplementary Display Module) file. This is a text file, constructed as described in the following sections:

- *SDM format guide*
- *SDM format reference* on page C-8.

### C.2.1 SDM format guide

Types are defined based on fundamental types, or composites of types.

Types can be parameterized, with the parameters acting as type-modifiers. All parameters are optional. All parameters are named, and parameters are passed in a named list of `(PARAMETER=VALUE, ...)`.

Types that can be user-visible have:

- a user-visible name
- an optional classification, a user-visible text string that assists the debug controller when organising a number of types. If a classification is specified, a name must be specified.

C++ style // comments are allowed, indicating that the rest of the line is a comment.

Two commands are defined, `INCLUDE` and `TYPEDEF`. These commands must appear at the beginning of a line, with no white space in front of them.

Except in definitions contained within an RDI register description file, such as `armperip.xml`, you can include other register type defintion files, using a `#include` structure:

```
INCLUDE “filename” // This is a comment.
```

The `TYPEDEF` command creates new types. It takes optional parameters of `CLASS` and `NAME`. If a type has a name, then it is user-visible. If it does not have a name, then it is visible only in other type definitions. Examples of types are `NUMERIC`, `FLAG`, `ENUM`, `COMPOSITE`, and `IEEE_FLOAT`. See *SDM format reference* on page C-8 for the definitive list.

For example,

```
TYPEDEF tZFLAG FLAG (SET="Z", UNSET="z") // This is a comment.
TYPEDEF tFIQ FLAG (SET="F", UNSET="f")   // So is this.
```

Type definitions of `ENUM` and `COMPOSITE` types require more information than only the type parameter list:

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

```
TYPEDEF tMODE ENUM (WIDTH=5, DEFAULT=”Reserved”)
{
    "User"=0x10,
    "FIQ"=0x11,
    "IRQ"=0x12,
    "SVC"=0x13,
    "Abort"=0x17,
    "Undef"=0x1b,
    "System"=0x1f}
```

Duplicate definitions are allowed in enums, to account for partially decoded enums. For example, a type given a user-visible name:

```
TYPEDEF tARMID (NAME=”Chip ID”, CLASS=”ARM”) ENUM (WIDTH=32)
{
    “ARM720T” = 0x41807200,
    “ARM740T” = 0x41807400,
    ...
}
```

In the case of composites, each atom is a `FIELD` or a `SEPARATOR`. A `FIELD` is defined as groups of bits:

```
    FIELD [hi:lo] {,[hi:lo] {...}} (NAME=”<name>”,
                                    TYPE=<type>{(<params>, ...)},
                                    ACCESS=”<access>”)
    SEPARATOR (NAME=“String”)
```

Atoms combine as follows, optionally grouped using braces to provide grouping as appropriate:

```
TYPEDEF tPACKEDFLOAT(NAME="2x32bit float", CLASS="Floating Point")
COMPOSITE (WIDTH=64)
{
    FIELD [63:32] (NAME="High", TYPE=IEEE_FLOAT (WIDTH=32)),
    FIELD [31:0]  (NAME="Low",  TYPE=IEEE_FLOAT (WIDTH=32))
}
TYPEDEF tPACKEDQ15 (NAME="2xQ-15-format", CLASS="DSP")  COMPOSITE (WIDTH=32)
{
    FIELD [31:16] (NAME="High", TYPE=QFORMAT (N=1,M=15)),
    FIELD [15:0]  (NAME="Low",  TYPE=QFORMAT (N=1,M=15))
}
TYPEDEF tPSR (NAME=”PSR”, CLASS=”ARM”) COMPOSITE (WIDTH=32)
{
    GROUP (NAME=”Flag bits”)
    {
        FIELD [31] (NAME="Zero Flag",     TYPE=tZFLAG, ACCESS=”RW”),
        FIELD [30] (NAME="Negative Flag", TYPE=tNFLAG, ACCESS=”RW”),
        FIELD [29] (NAME="Carry Flag",    TYPE=tCFLAG, ACCESS=”RW”),
        FIELD [28] (NAME="Overflow Flag", TYPE=tVFLAG, ACCESS=”RW”)
```


Copyright © 1999-2001 ARM Limited. All rights reserved.

```
    },
    FIELD [27:8] (TYPE=RESERVED (WIDTH=30), ACCESS=”0”),
    GROUP (NAME=”Mode bits”)
    {
        FIELD [7]  (NAME="Thumb bit",     TYPE=tTHUMB, ACCESS=”RW”),
        FIELD [6]  (NAME="IRQ bit",       TYPE=tIRQ,   ACCESS=”RW”),
        FIELD [5]  (NAME="FIQ bit",       TYPE=tFIQ,   ACCESS=”RW”),
        SEPARATOR (TEXTNAME=”_”)
        FIELD [4:0] (NAME="Mode",         TYPE=tMODE,  ACCESS=”RW”)
    }
}
```

The access parameter is interpreted as follows:

**Table C-7 Interpretation of access parameter**

| Value | Meaning |
|---|---|
| R | Readable |
| W | Writable |
| V | Reserved (write as read) |
| Z | Write as zero (read undefined) |
| 0 | Always 0 |
| 1 | Always 1 |
| U | Undefined |
| X | Not readable, but cacheing any written values is permitted and useful |
| N | Not cacheable (otherwise treated as cacheable) |


Copyright © 1999-2001 ARM Limited. All rights reserved.

### C.2.2 SDM format reference

Typedefs are constructed using the `TYPEDEF` type, described in Table C-8.

**Table C-8 Typedef type**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| `TYPEDEF RDIName (NAME, CLASS)` | - | - | `RDIName` = a unique name to define the type. |
| | | | `NAME` = “context name”. |
| | | | `CLASS` = “context group”. |

Fields are constructed using the `FIELD` type, described in Table C-9.

**Table C-9 Field type**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| `FIELD [hi:lo] {,[hi:lo] {...}} (NAME, TYPE, ACCESS)` | - | - | `NAME` = “text name”, for dialog and tooltip. |
| | | | `TYPE` = any predefined TYPEDEF or TYPEDEF already seen in current SDM text. |
| | | | `ACCESS` = “access modifier” (default = “RW”). |


Copyright © 1999-2001 ARM Limited. All rights reserved.

The fundamental types are described in Table C-10.

**Table C-10 Fundamental types, with parameters and classifications**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| NUMERIC (WIDTH, MIN, MAX, DEFAULT, PREFIX, PREPAD, PRINTF, TOOLTIP) | Numeric | General | WIDTH is in bits (in any type where it is used). Required. |
| | | | DEFAULT = “HEX”, “DEC”, “UDEC”, “OCT”, or “BIN” (default = “HEX”). |
| | | | PREFIX = “Y” or “N” (default dependent on DEFAULT). |
| | | | PREPAD = “Y” or “N” (default dependent on DEFAULT). |
| | | | PRINTF = any valid printf string for a numeric, for example “0x%08X” (default dependent on DEFAULT). If used, PREFIX and PREPAD are ignored. |
| | | | TOOLTIP = “A useful hint to the user”. |
| FLAG (SET, UNSET, TOOLTIP) | - | - | Implied WIDTH = 1, always. |
| | | | SET and UNSET are usually single character strings. |
| | | | TOOLTIP = “A useful hint to the user”. |
| ENUM (WIDTH, DEFAULT, TOOLTIP) | - | - | WIDTH = number of bits this enum represents. |
| | | | DEFAULT = “String to show when value not included in enumeration (not Selectable)”. |
| | | | TOOLTIP = “A useful hint to the user”. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table C-10 Fundamental types, with parameters and classifications (continued)**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| CHARACTER (WIDTH, PRINTF, TOOLTIP) | Character | General | Represents a WIDTH in bits, accepted values 7,8, N*8. Represents byte host-endian array of individual characters (not a nul-terminated string). |
| | | | PRINTF = any valid printf string for a character, for example “\”% 4s\”” (default dependent on WIDTH). |
| | | | TOOLTIP = “A useful hint to the user”. |
| IEEE_FLOAT (WIDTH, PRINTF, TOOLTIP) | IEEE Float | Floating point | WIDTH = 32 or 64, for now. |
| | | | PRINTF = any valid printf string for a float, for example “%f” (default dependent on WIDTH). |
| | | | TOOLTIP = “A useful hint to the user”. |
| FPA_SINGLE (PRINTF, TOOLTIP) | FPA Single | Floating point | WIDTH = 32, always. |
| | | | PRINTF = any valid printf string for a float, for example “%f” (default dependent on WIDTH). |
| | | | TOOLTIP = “A useful hint to the user”. |
| FPA_DOUBLE (PRINTF, TOOLTIP) | FPA Double | Floating point | WIDTH = 64, always. |
| | | | PRINTF = any valid printf string for a float, for example “%f” (default dependent on WIDTH). |
| | | | TOOLTIP = “A useful hint to the user”. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Table C-10 Fundamental types, with parameters and classifications (continued)**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| `FPA_EXTENDED (PRINTF, TOOLTIP)` | FPA Extended | Floating point | `WIDTH` = 80, always. |
| | | | `PRINTF` = any valid printf string for a float, for example "%f" (default dependent on `WIDTH`). |
| | | | `TOOLTIP` = "A useful hint to the user". |
| `FPA_INTERNAL (PRINTF, TOOLTIP)` | FP Internal | Floating point | To be confirmed. |
| `QFORMAT (N, M, PRINTF, TOOLTIP, DEFAULT)` | Q-format | DSP | `N` = numeric. |
| | | | `M` = numeric. |
| | | | `PRINTF` = any valid printf string for a float, for example "%f" (default dependent on `WIDTH`). |
| | | | `DEFAULT` = "UNSIGNED" (only for unsigned Q-format). |
| | | | `TOOLTIP` = "A useful hint to the user". |

Composite types are constructed using the composite type, described in Table C-11.

**Table C-11 Composite type**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| `COMPOSITE (WIDTH)` | - | - | `WIDTH` = number of bits expected in data. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

Group types are constructed using the group type, described in Table C-12 and only visible in a dialog.

**Table C-12 Group type**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| GROUP (NAME) | - | - | NAME = “Group box name”. |

Reserved types are constructed using the Reserved type, described in Table C-13.

**Table C-13 Reserved type**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| RESERVED (WIDTH, NAME, GUINAME, TEXTNAME) | - | - | WIDTH = number of bits expected in data. |
| | | | NAME = “Text to appear”. |
| | | | GUINAME = “Text to appear in GUI”. Overrides NAME. |
| | | | TEXTNAME = “Text to appear in monitor”. Overrides NAME. |

Separators contain no data and are constructed using the Separator type, described in Table C-14.

**Table C-14 Separator type**

| Type (param, param, ...) | Name | Classification | Notes |
|---|---|---|---|
| SEPARATOR (NAME, GUINAME, TEXTNAME) | - | - | NAME = “Text to appear”. |
| | | | If NAME = “NEWLINE” the dialog box forces a new line of controls from this point. |
| | | | GUINAME = “Text to appear in GUI”. Overrides NAME. |
| | | | TEXTNAME = “Text to appear in monitor”. Overrides NAME. |


Copyright © 1999-2001 ARM Limited. All rights reserved.

# Appendix D
# Using the Flash Downloader

This appendix describes the Flash downloader utility provided with ADS. It contains the following sections:


- *About the Flash downloader* on page D-2
- *Using the Flash downloader from AXD* on page D-4
- *Using the Flash downloader from armsd* on page D-5
- *Setting the IP address of a PID board* on page D-6.



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

## D.1 About the Flash downloader

The Flash downloader is a simple Flash utility that you can use to write a binary file to the Flash memory on an ARM Integrator board, or an ARM Development (PID) board. You can use the Flash downloader from the ADS debuggers, AXD and armsd.

The Flash downloader executes on the target board. When you invoke the Flash downloader from within a debugger, the debugger downloads the Flash downloader into RAM on the target board. The Flash downloader executes, and uses semihosting to fetch the code to program into Flash. The downloaded file must be in plain binary format. Refer to *ADS Linker and Utilities Guide* for information on converting an ELF format file to plain binary.

The Flash downloader requires either:

- Multi-ICE
- Angel, running from RAM.

### D.1.1 Integrator board version

The default Integrator version of the Flash downloader is supplied as a binary in `install_directory\bin\flash.li`. This can be used to program standard CFI-type Flash devices, for example the Intel DT28F320 and similar, as fitted to the ARM Integrator board.

**Note**

The Integrator version of the Flash downloader works only with the ARM Integrator board.

The ARM Integrator board cannot work in big-endian mode. A dummy `flash.bi` file is installed that issues a warning if you attempt to use it.

#### Setting the Integrator board configuration switches

The switch settings on the Integrator board select whether the default image, the boot monitor, or a user image is run on reset.

The sequence below works for downloading to most Integrator boards:

1. Set switch 1 to on.

   Refer to the manuals provided with the Integrator board for more details on settings.

2. Turn the board power off then back on.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

3. Start AXD and use the Flash downloader.
4. Set switch 1 to off.
5. Turn the board power off then back on to run the downloaded image.

If you load and run an image that does not do ROM/RAM remapping, subsequent attempts to load or run any other image fail with an undefined instruction error. Use the boot monitor in ROM on the Integrator board and a terminal emulator to clear the Flash.

### D.1.2 PID board version

Big-endian and little-endian versions of the Flash downloader for the ARM Development (PID) board are supplied in:

- *install_directory*\bin\flashpid.li
- *install_directory*\bin\flashpid.bi.

To use the PID version of the little-endian Flash downloader from AXD, rename `flash.li` to `flash_Integrator.li` (or similar), and rename `flashpid.li` to `flash.li`.

To use the PID version of the big-endian Flash downloader from AXD, rename `flash.bi` to `flash_dummy.bi` (or similar), and rename `flashpid.bi` to `flash.bi`.

**Note**

The PID versions of the Flash downloader fail if they do not recognize the Flash memory being used. The PID versions of the Flash downloader recognize the two Flash devices supported by the ARM Development (PID) board, the ATMEL AT29C040A (4 Mbit, 8-bit) and AT29C1024 (1 Mbit, 16-bit) Flash devices.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## D.2 Using the Flash downloader from AXD

Follow these steps to use the Flash downloader from AXD:

1. Select **Flash Download…** from the **File** menu. The Flash DownLoad dialog is displayed (Figure D-1).

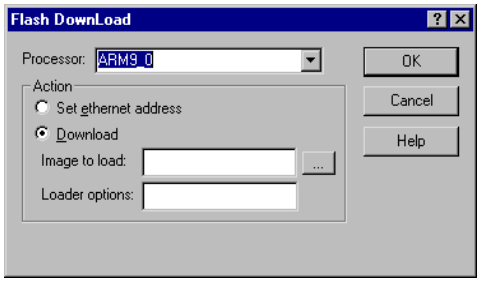



**Figure D-1 Flash DownLoad dialog**

2. Specify the input information or click **Browse** to select a binary file to download. You can either use the default block, image, and address values or enter new values:
   - If you do not enter any loader information, the downloader uses the default values:

     **Image number** 128

     **Block number** 0

     **Image base** `0x24000000`
   - If you require different values than the defaults, enter the input information in the format:

     `[a<address> |or| b<block_no>] i<image_no> pathname`

     For example:

     `b5 i5 my_image`

     **Note**

     If the pathname to the binary file contains spaces you must enclose the pathname in quotes.

3. Click **OK**. The Flash downloader reads the binary file and displays the download settings in the Console processor view. You can edit the settings if required.

4. Edit the settings, if required, or press Enter. The Console view displays a message when the Flash is written.


Copyright © 1999-2001 ARM Limited. All rights reserved.

## D.3 Using the Flash downloader from armsd

**Note**

This section applies only if you are targeting Angel or EmbeddedICE. armsd does not support Multi-ICE.

To use the Flash downloader from the command line (assuming that you have a serial/parallel connection) write a batch file containing this command:

```
armsd -adp -port s=1,p=1 -line 38400 -exec flash ROMname
```

where:

*flash* Is the name of the Flash downloader. By default this is: *install_directory*\bin\flash.li for the ARM Integrator board.

**Note**

If you want to use the Flash downloader for the ARM Development (PID) board, you must specify the actual file name as a parameter to armsd using, for example:

```
armsd -adp -port s=1,p=1 -line 38400 -exec flashpid.li ROMname
```

*ROMname* Is the name of the binary file that you want to be programmed into Flash memory.

**Note**

If the pathname to the binary file contains spaces you must enclose the pathname in quotes.

Execute the batch file to download to Flash. Enter the address to start writing from when prompted to do so.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

## D.4 Setting the IP address of a PID board

If you are using the Angel Ethernet Kit with an ARM Development (PID) board, you can use the Flash downloader program to override the default IP address and net mask used by Angel for Ethernet communication:

- From AXD, select the **Set ethernet address** button in the Flash DownLoad dialog (see Figure D-1 on page D-4).
- From armsd, pass the Flash download program the argument `-e`. The program prompts for the IP address and net mask.

**Note**

The Ethernet option is not applicable to the ARM Integrator board, and is ignored.


Copyright © 1999-2001 ARM Limited. All rights reserved.

# Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

**Action Point**
A breakpoint or watchpoint (see *Breakpoint* and *Watchpoint*), at which a specified debugging action occurs. The default action is to stop execution. Another typical action you can specify is to record a diagnostic message in a log file and continue execution.

**ADP**
See *Angel Debug Protocol*.

**ADS**
See *ARM Developer Suite*.

**Angel**
Angel is a debug monitor that enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either *ARM state* or *Thumb state*.

**Angel Debug Protocol**
Angel uses a debugging protocol called the *Angel Debug Protocol* (ADP) to communicate between the host system and the target system. ADP supports multiple channels and provides an error-correcting communications protocol.

**ARM Developer Suite**
A suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of *RISC* processors.


*Copyright © 1999-2001 ARM Limited. All rights reserved.*

**ARM eXtended Debugger**
The *ARM eXtended Debugger* (AXD) is the latest debugger software from ARM that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. AXD is supplied in both Windows and UNIX versions.

**ARM state**
A processor that is executing ARM (32-bit) instructions is operating in ARM state (see also *Jazelle state* and *Thumb state*).

**ARM symbolic debugger**
*ARM Symbolic Debugger* (armsd) is an interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language. It is a command-line debugger that runs on all supported platforms.

**armsd**
See *ARM Symbolic Debugger*.

**ARMulator**
ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.

**ATPCS**
ARM/Thumb Procedure Call Standard.

**AXD**
See *ARM eXtended Debugger*.

**Big-endian**
Memory organization where the least significant byte of a word is at a higher address than the most significant byte. See also *Little-endian*.

**Breakpoint**
A location in the image. If execution reaches this location, the debugger halts execution of the image. See also *Watchpoint*.

**ByteCode**
ByteCode format specifies the use of Jazelle bytecodes which are platform-independent instructions, generated by a compiler, and run on the Java Virtual Machine (JVM).

**Class**
A C++ class involved in the image.

**Class variables/functions**
Variables or functions with scope limited to the current class. (See also *Local variables/functions* and *Global variables/functions*.)

**CLI**
See *Command-line Interface*.

**Command-line Interface**
You can operate any ARM debugger by issuing commands in response to command-line prompts. This is the only way of operating armsd, but AXD offers a graphical user interface in addition. A command-line interface is particularly useful when you need to run the same sequence of commands repeatedly. You can store the commands in a file and submit that file to the command-line interface of the debugger.

**Context**
The information stored in a block of registers on entry to a subroutine, and held there until needed for restoring the information on exit from the subroutine.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

**Context menu** See *Pop-up menu*.

**Control Bars** A control bar is a special window which is usually aligned along one side of a frame window. Control bars can be considered containers for other windows and controls or as a drawing area for the application.

**Coprocessor** An additional processor used for certain operations. Usually used for floating-point calculations, signal processing, or memory management.

**CPSR** Current Program Status Register. See *Program Status Register*.

**DCC** See *Debug Communications Channel*.

**Debug Communications Channel**
A debug communications channel allows data to be passed between the target and the host debugger using the JTAG port and an EmbeddedICE interface, without stopping the program flow or entering debug state.

**Debugger** An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.

**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.

**DLL** See *Dynamic Linked Library*.

**Dockable Windows** A dockable window is positioned and sized automatically when you open it or dock it, with any other docked windows already on the screen being resized if necessary. You can change the size of a docked window, or undock it and allow it to float free on the desktop.

**Double word** A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

**DWARF** Debug With Arbitrary Record Format.

**Dynamic Linked Library**
A collection of programs, any of which can be called when needed by an executing program. A small program that helps a larger program communicate with a device such as a printer or keyboard is often packaged as a DLL.

**ELF** Executable and Linking Format.

**Enhanced Program Status Register**
See *Program Status Register*.

**EPSR** Enhanced Program Status Register. See *Program Status Register*.

**Executable image** See *Image*.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

**File**
A disk file somehow involved in the debuggee or debugger. This will most likely be a source file compiled/assembled into an image. However it might also be an image file or a session file.

**Flash downloader**
The Flash downloader is used to download binary images to the Flash memory of supported ARM development boards.

**Floating point**
Convention used to represent real (as opposed to integer) numeric values. Several such conventions exist, trading storage space required against numerical precision.

**Floating point emulator**
Software that emulates the action of a hardware unit dedicated to performing arithmetic operations on floating-point values.

**FP**
See *Floating point*.

**FPE**
See *Floating Point Emulator*.

**Function**
A C++ method or free function.

**Global variables/functions**
Variables or functions with global scope within the image. (See also *Class variables/functions* and *Local variables/functions*.)

**Harvard architecture**
A processor architecture incorporating physically separate memories and associated buses for holding instructions and data.

**Halfword**
A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

**Host**
A computer which provides data and other services to another computer, or a computer that has applications programs installed and available for use.

**ICE**
In-Circuit Emulator.

**IDE**
See *Integrated Development Environment*.

**Image**
A file of executable code which can be loaded into memory on a target and executed by a processor there.

**Integrated development environment**
An IDE provides facilities for automating image-building and file-management processes, for example the CodeWarrior IDE in ADS.

**Jazelle**
ARM's technology for Java applications that enables Jazelle-capable processors, such as the ARM926EJ-S, to execute Java bytecode directly.

**Jazelle state**
A processor that is executing Jazelle bytecode (8-bit) instructions is operating in Jazelle state (see also *ARM state* and *Thumb state*).


*Copyright © 1999-2001 ARM Limited. All rights reserved.*

**Joint Test Action Group**
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).

**JPSR** Jazelle Program Status Register. See *Program Status Register*.

**JTAG** See *Joint Test Action Group*.

**Little-endian** Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also *Big-endian*.

**Local variables/functions**
Variables or functions with local scope. (See also *Class variables/functions* and *Global variables/functions*.)

**MDI** See *Multiple Document Interface*.

**Memory management unit**
Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.

**MMU** See *Memory Management Unit*.

**Multi-ICE** Multi-processor based JTAG debug tool for embedded systems. ARM registered trademark.

**Multiple Document Interface**
A feature of MS Windows allowing the simultaneous display of a number of windows.

**von Neumann architecture** A processor architecture that does not distinguish between memory that holds instructions and memory that holds data.

**PID** A platform-independent development board designed and supplied by ARM Ltd.

**Pop-up menu** Also known as *Context menu*. A menu that is displayed temporarily, offering items relevant to your current situation. Obtainable in most ADS windows 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** An actual processor, real or simulated running on the target. A processor always has at least one context of execution.

**Processor Status Register**
See *Program Status Register*.

**Profiling** Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

*Call-graph profiling* provides great detail but slows execution significantly. *Flat profiling* provides simpler statistics with less impact on execution speed.

For both types of profiling you can specify the time interval between statistics-collecting operations.

**Program image**
See *Image*.

**Program Status Register**
*Program Status Register* (PSR), containing some information about the current program and some information about the current processor.

Is also referred to as *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR). The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned.

An *Enhanced Program Status Register* (EPSR) contains an additional bit (the Q bit, signifying saturation) used by some ARM processors, including the ARM9E.

A *Jazelle Program Status Register* (JPSR) contains an additional bit (the J bit, signifying Jazelle state) used by some ARM processors, including the ARM9EJ-S.

**PSR**
See *Program Status Register*.

**RDI**
See *Remote Debug Interface*.

**Register**
A processor register.

**Remote_A**
A communications protocol used, for example, between debugger software such as *ARM eXtended Debugger* (AXD) and a debug agent such as *Angel*.

**Remote Debug Interface**
The *Remote Debug Interface* (RDI) is an ARM standard procedural interface between a debugger and the debug agent. RDI gives the debugger a uniform way to communicate with:

- a debug agent 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).

**Saved Program Status Register**
See *Program Status Register*.

**Scope**
The range within which it is valid to access such items as a variable or a function. See also *Class, Global* and *Local variables/functions*.


Copyright © 1999-2001 ARM Limited. All rights reserved.

**Script**
A file specifying a sequence of debugger commands that you can submit to the command-line interface using the `obey` command. This saves you from having to enter the commands individually, and is particularly helpful when you need to issue a sequence of commands repeatedly.

**SDT**
See *Software Development Toolkit*.

**Semihosting**
A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.

**Software Development Toolkit**
*Software Development Toolkit* (SDT) is an ARM product still supported but superseded by *ARM Developer Suite* (ADS).

**Source File**
A file which is processed as part of the image building process. Source files are associated with images.

**SPSR**
Saved Program Status Register. See *Program Status Register*.

**Stack backtracing**
Examining the list of currently active subroutines in a halted executing program to help establish how current settings have arisen.

**Tabbed**
A GUI mechanism to overlay several pages in a single window, allowing page selection by clicking on a named tab.

**Target**
The target processor (real or simulated), on which the target application is running.

The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.

**Thumb state**
A processor that is executing Thumb (16-bit) instructions is operating in Thumb state (see also *ARM state* and *Jazelle state*).

**Tracing**
Recording diagnostic messages in a log file, to show the frequency and order of execution of parts of the image. The text strings recorded are those that you specify when defining a breakpoint or watchpoint. See *Breakpoint* and *Watchpoint*. See also *Stack backtracing*.

**Vector Floating Point**
A standard for floating-point coprocessors where several data values can be processed by a single instruction.

**VFP**
See *Vector Floating Point*.

**Views**
Windows showing the data associated with a particular debugger/target object. These might consist of a single, simple GUI control such as an edit field or a more complex multi-control dialog implemented as an ActiveX.


Copyright © 1999-2001 ARM Limited. All rights reserved.

The **Processor Views** menu allows you to select views associated with a specific processor, while the **System Views** menu allows you to select system-wide views.

**Watchpoint**
A location in the image that is monitored. If the value stored there changes, the debugger halts execution of the image. See also *Breakpoint*.

**Word**
A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.


*Copyright © 1999-2001 ARM Limited. All rights reserved.*

# Index

## A


About this book viii
Accelerator keys 2-14
Access protection, in
  AXD expressions 4-5
Accessing
  host peripherals 1-9
  online help 1-12, 9-22
Address of DCC semihosting SWI
    handler 4-11
Addresses, entering 4-12
Agent, debug 1-2
Alias armsd command 9-10
Analysis of processor time 4-27
Angel 1-9
  configuring 5-94
  Debug Protocol (ADP) 2-8, 7-4
Applying for a software license 2-2
Arguments armsd command 9-11
Arguments, command-line 2-3, 5-52,
    7-3
ARM
  disassembly mode 5-80
ARM ADI 1-9
ARM Agilent Debug Interface 1-9
ARM core 1-8
ARM debuggers 1-11
  armsd 7-2
  AXD 2-1
armsd 7-1
  address constants 8-6
  armsd.ini file 7-2
  ARMulator 7-4
  backtrace 9-3
  big-endian memory 7-3
  breakpoints 8-4, 9-2
  character constants 8-6
  clock speed 7-4
  command-line arguments for
    debugee 9-25
  command-line options 7-3
  communications channel 9-46
  configuring 7-2
  constants 8-6
  context of variables 8-2
  coprocessor register display 9-4
  displaydesc argument 9-15
  duration of simulation 8-7
  echoing commands 8-7
  EmbeddedICE commands 9-47
  EmbeddedICE variables 8-10
  execution options 7-4
  expressions as arguments 8-5
  flash downloader D-5
  floating point emulator 7-3
  formatting output 8-11
  getting started 8-1
  help on 7-3
  high-level language variables 8-2
  high-level languages 9-23
  initialization file 7-2
  input from named file 7-4
  internal variables 8-10
  invoking 7-3
  let command 8-11
  list of variables 8-7
  little-endian memory 7-3
  loading debug information 7-4



*Copyright © 1999-2001 ARM Limited. All rights reserved.*

low-level debugging 8-13, 9-3
low-level symbols 8-13, 8-15
Multi-ICE variables 8-10
multi-statement lines 9-45
names of variables 8-2
operating system commands 9-5
output to file 7-4
overview 7-2
predefined symbols 8-14
print command 8-11
procedure names 8-4
processor type 7-3
profiling data 9-5
program line numbers 8-4
program locations 8-4
prompts 9-5, 9-6
remote debugging using ADP 7-4
search paths 7-4
setting the psr 8-15
source-level objects 8-2
starting debugee 9-3
starting debugger 7-3
statements withina line 8-4
stopping debugee 9-2
stopping debugger 9-5, 9-35
subscripts, pointers and arrays 8-6
symbols 7-4
syntax overview 7-3
variables 8-7
watchpoints 9-3
armsd commands 9-7, A-2
alias 9-5, 9-10
arguments 9-2, 9-11
backtrace 9-3, 9-11
break 9-2, 9-12
call 9-2, 9-13
ccin 9-46
ccout 9-46
comment 9-5, 9-9
compared with AXD A-2
context 9-3, 9-16
coproc 9-4, 9-13
cregdef 9-4, 9-16
cregisters 9-4, 9-16
cwrite 9-4, 9-17
examine 9-4, 9-17
find 9-4, 9-18
fpregisters 9-4, 9-19
getfile 9-3, 9-21
go 9-3, 9-21
help 9-5, 9-22
in 9-3, 9-22
istep 9-3, 9-22
language 9-3, 9-23
let 8-15, 9-24
list 9-4, 9-27
listconfig 9-47
load 9-3, 9-28
loadagent 9-47
loadconfig 9-47
localvar 9-28
log 9-5, 9-30
lsym 9-4, 9-31
obey 9-5, 9-31
out 9-3, 9-32
pause 9-5, 9-6, 9-32
print 9-5, 9-33
profclear 9-5, 9-6, 9-33
profoff 9-5, 9-6, 9-33
profon 9-5, 9-6, 9-34
profwrite 9-5, 9-34
putfile 9-3, 9-35
quit 9-5, 9-35
readsyms 9-36
registers 9-4, 9-36
reload 9-3, 9-37
selectconfig 9-48
step 9-3, 9-39
symbols 9-40
type 9-5, 9-41
unbreak 9-3, 9-42
unwatch 9-3, 9-42
variable 9-2, 9-43
watch 9-3, 9-43
where 9-3, 9-44
while 9-45
! 9-5, 9-9
| 9-5, 9-9
ARMulate.cnf file 2-6
ARMulator 1-8
configuring 2-6, 5-88
floating point emulator 5-90
Array expansion 2-10, 5-84
ASCII
format 4-19
search string 5-17
ASIC 1-8
Asm, specifying in CLI 6-9
Assembly code interleaved with C++ 5-44
Audience, intended viii
AXD
CLI window 6-2
closing down 2-5, 5-15
command-line operation 5-65, 6-2
commands 6-13
configuring 5-80
desktop 5-2
displays 2-9
execute menu 5-76
file menu 5-6
flash download 5-11
help menu 5-102
menu bar 5-2
menus 2-12, 5-2
options menu 5-80
processor views menu 5-18
search menu 5-16
starting 2-3
status bar 5-5, 5-98
system views menu 5-48
toolbars 5-3
tools 2-14
window menu 5-99
AXD CLI commands 6-13, A-2
backtrace 6-13
break 6-14
cclasses 6-16
cfunctions 6-16
classes 6-16
clear 6-17
clearbreak 6-17
clearstat 6-18
clearwatch 6-19
comment 6-19
compared with armsd A-2
context 6-20
convariables 6-21
cvariables 6-22
dbginternals 6-22
disassemble 6-23
echo 6-24
examine 6-24
files 6-25
fillmem 6-25
findstring 6-26
findvalue 6-27


 Copyright © 1999-2001 ARM Limited. All rights reserved. 

format 6-28
functions 6-29
getfile 6-29
go 6-29
help 6-29
images 6-30
imgproperties 6-30
importformat 6-31
let 6-31
list 6-31
listformat 6-32
load 6-32
loadbinary 6-33
loadsession 6-34
loadsymbols 6-34
log 6-34
lowlevel 6-35
memory 6-35
obey 6-36
parse 6-36
print 6-36
processors 6-37
procproperties 6-37
putfile 6-37
quitdebugger 6-38
readsyms 6-38
record 6-38
regbanks 6-38
registers 6-39
reload 6-40
run 6-40
runtopos 6-41
savebinary 6-42
savesession 6-43
setaci 6-43
setbreakprops 6-44
setimgprop 6-45
setmem 6-45
setpc 6-46
setproc 6-46
setprocprop 6-47
setreg 6-48
setsourcedir 6-49
setwatch 6-50
setwatchprops 6-51
source 6-52
sourcedir 6-53
stackentries 6-53
stackin 6-54
stackout 6-54
statistics 6-54
step 6-55
stepsize 6-56
stop 6-56
trace 6-56
traceload 6-57
type 6-57
unbreak 6-57
unwatch 6-57
update 6-57
variables 6-58
watch 6-59
watchpt 6-59
where 6-61
AXD processor views 5-18
backtrace 5-29
console 5-39
debug comms channel 5-37
disassembly 5-40
low-level symbols 5-35
memory 5-31
registers 5-19
source... 5-44
variables 5-26
watch 5-23
AXD system views 5-48
command-line interface 5-65
control 5-49
debugger internals 5-69
output 5-63
registers 5-54
watch 5-56

## B

Backtrace
AXD view 5-29
Backtrace armsd command 9-11
Backtrace AXD command 6-13
Base classes
in AXD 4-4
Binary format 4-19
Books, related x
Book, about this viii
Break armsd command 9-12
Break AXD command 6-14
Breakpoints
deleting 5-79
identifying in CLI 6-9
in armsd 8-4, 9-2
in AXD 3-4, 5-58, 5-78

## C

Call armsd command 9-13
Cclasses AXD command 6-16
C-cycles 5-73
Cfunctions AXD command 6-16
Changing values of variables
in AXD 5-26
Class
page in Control view 5-50
Classes AXD command 6-16
Class, identifying in CLI 6-9
Clear AXD command 6-17
Clearbreak AXD command 6-17, 6-18
Clearwatch AXD command 6-19
Closing
armsd 9-35
AXD 2-5
Code, ARM/Thumb/Jazelle 5-80, 6-6
Command-line arguments for debugee
in armsd 7-3
in AXD 5-52
Command-line arguments for debugger
armsd 7-3
AXD 2-3
Command-line operation
definitions 6-9
of armsd 9-7, A-2
of AXD 5-65, 6-2, A-2
predefined parameters 6-6
Commands
AXD and armsd compared A-2
echoing 8-7
in armsd 9-1
in AXD 6-13
that use lists 6-5
Comment AXD command 6-19
Comments
in script files 9-9
on ADS xii
on documentation xii
Comms channel 4-11, 5-37, 5-93, 5-96, 5-97, 6-10, 9-46

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

AXD view 5-37
Concepts 1-2
Configuring
armsd debugger 7-2
ARMulator 2-6, 5-88
AXD debugger 5-80
debugger target 5-87
Multi-ICE 5-93
Remote_A 5-94
Console
AXD view 5-39
Constants 8-6
Context armsd command 9-16
Context AXD command 6-20
Context of execution 5-78
Context of program 1-3, 6-54, 9-22, 9-32
Context, identifying in CLI 6-9
Control AXD view 5-49
Convariables AXD command 6-21
Coproc armsd command 9-13
Coprocessor register
changing contents 9-17
descriptions B-1
displaying contents 9-16
Cregdef armsd command 9-16
Cregisters armsd command 9-16
Cvariables AXD command 6-22
Cwrite armsd command 9-17
Cycle counts 5-73
C++
interleaved with assembly code 5-44


## D


Data entry formats 4-12, 4-16
Dbginternals AXD command 6-22
Debug
agent 1-2
comms channel SWI handler 4-11
comms channel viewing 4-11, 5-37, 5-93, 5-96, 5-97, 6-10, 9-46
log 5-63
monitor 1-9
protocol (ADP) 2-8, 7-4
session, restoring 5-12
symbols 5-8, 5-29
typical setup 1-6
Debugger internals
AXD view 5-69
Debuggers
closing down 2-5, 9-35
currently supported 1-11
starting program execution 5-76, 9-21
starting up 2-3, 7-3
Decimal 4-12
Decimal format 4-18
Deleting breakpoints
in armsd 9-42
in AXD 5-58, 5-79
Demonstration programs 3-2
Desktop
AXD 5-2
Development board 1-6
Disassemble AXD command 6-23
Disassembly
AXD view 5-40
mode 5-42, 5-80
sequence break 5-45
Display formats 4-16
Displaying interleaved code 5-44
Documentation feedback xii
Download
flash D-2


## E


Echo AXD command 6-24
Editing breakpoints
in AXD 5-58
EmbeddedICE 1-8
Enquiries xii
Entering addresses 4-12
E-PSR, setting, in AXD 5-20
Examine armsd command 9-17
Examine AXD command 6-24
Examining
memory 5-31
source files 5-9, 5-16, 5-44
variables 5-26
Examples 3-2
breakpoint setting 3-4
changing memory contents 3-16
examining memory 3-14
examining registers 3-12
examining variables 3-8
updating a program 3-18
watchpoint setting 3-6
Exceptions intercepted 5-97
Execute AXD menu 5-76
Execution
context 5-78
starting 5-76, 9-21
stopping 5-77
stopping and stepping 4-2
Exiting debugger 2-5, 5-15, 9-35
Expansion of arrays 2-10, 5-84
Expressions
as arguments 8-5
guidelines for using 4-4
sample 4-5
specifying 4-4
watching, in AXD 5-57


## F


Feedback
on ADS xii
on documentation xii
File AXD menu 5-6
Files
armsd.ini 7-2
ARMulate.cnf 2-6
AXD command 6-25
identifying, in CLI 6-9
page in Control view 5-50
recently-opened 5-13
Fillmem AXD command 6-25
Find armsd command 9-18
Findstring AXD command 6-26
Findvalue AXD command 6-27
Flash download
override IP address and net mask D-6
Flash downloader D-2
Floating point
emulator, in armsd 7-3
emulator, in ARMulator 5-90
formats 4-20
Floating-point
returning values from armsd 8-8
vector 5-70, 8-9


 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Format
  ASCII 4-19
  AXD command 6-28
  binary 4-19
  decimal 4-18
  display 4-16, 5-27, 5-80
  floating point 4-20
  for data entry 4-16
  hexadecimal 4-18, 4-26
  octal 4-18, 4-26
  of armsd output 8-11
  printf 4-19
  Q-format 4-24
  registers 4-21
  scientific 4-20
  specifying in CLI 6-9
  string 4-25
  submenus 4-16
  U decimal 4-25
Fpregisters armsd command 9-19
Function calls to RDI 5-63
Functions
  AXD command 6-29
  stepping into/out of 5-77, 6-7

## G

Getfile armsd command 9-21
Getfile AXD command 6-29
Glossary Glossary-1
Go armsd command 9-21
Go AXD command 6-29

## H

Halfwords, reading/writing
  in armsd 9-25
Help armsd command 9-22
Help AXD command 6-29
Help, online 1-12, 5-102, 9-22
Hexadecimal 4-12
  format 4-18, 4-26
  search string 5-17
High-level languages and armsd 8-2
High-level symbols 4-10, 4-12
Host peripherals, accessing 1-9

## I

I-cycles 5-73
Image
  loading 5-7, 6-32, 9-28
  page in Control view 5-49
  reloading 5-9, 6-40, 9-37
  stepping through 5-77, 6-55, 9-39
  stopping execution of 5-77
Images AXD command 6-30
Images, recently-opened 5-13
Image, identifying in CLI 6-9
Imgproperties AXD command 6-30
Importformat AXD command 6-31
In armsd command 9-22
Indicators
  # 6-4, 8-14, 9-7
  $ 9-7
  + 6-4
  @ 4-10, 6-4, 9-7
  ^ 8-13, 9-7
  | 6-4
Instr, specifying in CLI 6-10
Intended audience viii
Intercepted exceptions 5-97
Interfacing with targets 1-5
Interleaving C++ and assembly code 5-44
Internal variables 5-69, 8-7
IPvariable, specifying in CLI 6-10
Istep armsd command 9-22

## J

Jazelle
  disassembly mode 5-80
J-PSR, setting, in AXD 5-20
JTAG 1-8

## K

Keyboard shortcuts 2-14

## L

Language armsd command 9-23
Languages
  high-level and armsd 8-2
Launching AXD from DOS 2-3
Let
  armsd command 8-11
  AXD command 6-31
Let armsd command 9-24
License application 2-2
License-managed software 2-2
Line number 4-12
Line numbers
  in programs 8-4
Line, stepping to next 5-77, 6-55
List armsd command 9-27
List AXD command 6-31
Listformat AXD command 6-32
Lists in AXD commands 6-5
Load armsd command 9-28
Load AXD command 6-32
Load session 5-12
Loadbinary AXD command 6-33
Loading
  an image 5-7, 6-32, 9-28
  debug symbols 5-8
  memory from file 5-9
Loadsession AXD command 6-34
Loadsymbols AXD command 6-34
Localvar armsd command 9-28
Log armsd command 9-30
Log AXD command 6-34
Log messages 5-63
Logging CLI input and output 5-66
Lowlevel AXD command 6-35
Low-level debugging in armsd 8-13
Low-level symbols 4-10, 4-12
  AXD view 5-35
Lsym armsd command 9-31

## M

Member functions, in
  AXD expressions 4-5
Memory
  AXD command 6-35
  AXD view 5-31
  loading from file 5-9
  modifying 3-16
  reading/writing, in armsd 9-25

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

reading/writing, in AXD 5-35
saving to file 5-10
viewing 3-14
word size for display in CLI 6-10
Menu bar
AXD 5-2
Menus
AXD 2-12, 5-2
AXD view-specific 2-13, 5-2
pop-up 2-13
Mode
disassembly 5-42, 5-80
stepping 5-42, 5-46
Modifying variables 5-26
Monitor, debug 1-9
Multi-ICE 1-8, 5-93


## N


N-cycles 5-73


## O


Obey armsd command 9-31
Obey AXD command 6-36
Octal format 4-18, 4-26
Online help 1-12, 9-22
Opening a source file 5-9
Operating system
accessing from armsd 9-9
Operators, in
AXD expressions 4-4
Options AXD menu 5-80
Out armsd command 9-32
Output AXD view 5-63


## P


Parameters, predefined, for CLI 6-6
Parse AXD command 6-36
Paths, search 5-98
Pause armsd command 9-32
Peripherals, accessing 1-9
Persistence 4-13
Pop-up menus 2-13
Position, identifying in CLI 6-10
PPvariable, specifying in CLI 6-10
Print armsd command 8-11, 9-33
Print AXD command 6-36
Printf format 4-19
Problem solving xii
Procedure names 8-4
Procedures, tutorial 3-2
Processor time analysis 4-27
Processor Views AXD menu 5-18
Processors AXD command 6-37
Processors, identifying in CLI 6-11
Processors, simulated 1-8
Procproperties AXD command 6-37
Product feedback xii
Profclear armsd command 9-33
Profiling 4-27, 5-98, 9-34
interval 5-7, 5-52, 9-34
Profoff armsd command 9-33
Profon armsd command 9-34
Profwrite armsd command 9-34
Program
context 6-54, 9-22, 9-32
demonstration 3-2
executing in armsd 9-21
line numbers 8-4
locations 8-4
reloading 5-9
stopping and stepping 4-2
updating 3-18
Protocol, Angel Debug (ADP) 2-8, 7-4
PSR, setting
in armsd 8-15
in AXD 5-20
Publications, related x
Putfile armsd command 9-35
Putfile AXD command 6-37


## Q


Q-format 4-24
Queries xii
Quitarmsd command 9-35
Quitdebugger AXD command 6-38
Quitting
armsd 9-35
AXD 2-5, 5-15


## R


RDI function calls 5-63
RDI (Remote Debug Interface) 1-3
Readsyms armsd command 9-36
Readsyms AXD command 6-38
Recent sessions 5-14
Recently-opened
files 5-13
images 5-13
symbols files 5-14
Record AXD command 6-38
Regbanks AXD command 6-38
Regbank, identifying in CLI 6-11
Registers
adding to system view 5-21, 5-55
AXD command 6-39
AXD processor view 5-19
AXD system view 5-54
changing values of 5-20
coprocessor B-1
demonstration of viewing 3-12
formats 4-21
halt if changed 5-61
specifying in CLI 6-12
Registers armsd command 9-36
Related publications x
Reload armsd command 9-37
Reload AXD command 6-40
Reloading an image 5-9, 6-40, 9-37
Remote Debug Interface 1-3
Remote_A 1-9
configuring 5-94
Restoring a debug session 5-12
Run AXD command 6-40
Run to cursor 5-78
Running a demonstration program 3-2
Runtopos AXD command 6-41


## S


Save session 5-13
Savebinary AXD command 6-42
Savesession AXD command 6-43, 6-51
Saving a debug session
Debug
session, saving 5-13


 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Saving memory to file 5-10
Scope
  of variables 1-4
  specifying in CLI 6-12
S-cycles 5-73
Search AXD menu 5-16
Semihosting mode 4-11, 5-97
Session
  loading 5-12
  saving 5-13
Sessions, recent 5-14
Set watchpoint 5-79
Setaci AXD command 6-43
Setbreakprops AXD command 6-44
Setimgprop AXD command 6-45
Setmem AXD command 6-45
Setpc AXD command 6-46
Setproc AXD command 6-46
Setprocprop AXD command 6-47
Setreg AXD command 6-48
Setsourcedir AXD command 6-49
Setting up targets 2-6
Setup, typical 1-6
Setwatch AXD command 6-50
Shortcut keys 2-14
Simulation
  duration of 8-7
  of processors 1-8
Software, license-managed 2-2
Source AXD command 6-52
Source files
  examining 5-9
Source path, specifying 5-98
Sourcedir AXD command 6-53
Source... AXD view 5-44
stack
  broken 5-29
Stackentries AXD command 6-53
Stackin AXD command 6-54
Stackout AXD command 6-54
Starting
  armsd 7-3
  AXD 2-3
Statements within a line 8-4
Statistics 5-72
Statistics AXD command 6-54
Status bar
  in AXD 5-5, 5-98
Step armsd command 9-39
Step AXD command 6-55
Stepping
  mode 5-42, 5-46
  through an image 4-2, 6-55, 9-39
  through assembler code 4-3, 9-32, 9-39
Stepsize AXD command 6-56
Step, specifying in CLI 6-12
Stop AXD command 6-56
Stopping
  armsd 9-35
  AXD 2-5, 5-15
  execution of image 5-77
String
  format 4-25
Strings
  specifying in CLI 6-12
Structure of this book viii
Subscripts, pointers and arrays 8-6
Symbolic debugger (armsd) 7-1
Symbols
  debug 5-8, 5-29
  files, recently-opened 5-14
  high- and low-level 4-10, 4-12
  out-of-sequence symbol 5-45
Symbols armsd command 9-40
System Views AXD menu 5-48

## T

Target
  configuring 5-87
  interfacing with 1-5
  page in Control view 5-49
  setting up 2-6
  variables 5-71
Terminology 1-2, Glossary-1
Thumb
  breakpoint setting 5-60
  channel viewer 5-37
  disassembly mode 5-80
Time analysis 4-27
Toggle breakpoint 5-78
Toggle parameter, in CLI 6-12
Toggle watchpoint 5-79
Toolbar
  AXD 5-3
Tools, AXD 2-14
Trace AXD command 6-56
Traceload AXD command 6-57
Tutorial examples 3-2
Type armsd command 9-41
Type AXD command 6-57

## U

Unbreak armsd command 9-42
Unbreak AXD command 6-57
Unwatch armsd command 9-42
Unwatch AXD command 6-57
Update AXD command 6-57

## V

Values, specifying in CLI 6-12
Variable armsd command 9-43
Variables
  AXD command 6-58
  AXD view 5-26
  changing contents of, in AXD 5-26
  demonstration of viewing 3-8
  halt if changed 5-61
  in armsd 8-7
  in specificfunction activation 8-3
  referenced from armsd 8-2
  scope of 1-4
  target-specific 5-71
  watching, in AXD 5-23
Vector catch variable in AXD 5-97
Vector floating point (VFP) 5-70, 8-9

## W

Watch
  AXD command 6-59
  AXD processor view 5-23
  AXD system view 5-56
Watch armsd command 9-43
Watchpoints
  clearing in armsd 9-42
  identifying in CLI 6-12
  in AXD 3-6, 5-61, 5-79
  setting 9-43
Watchpt AXD command 6-59

 Copyright © 1999-2001 ARM Limited. All rights reserved. 

Where armsd command 9-44
Where AXD command 6-61
While armsd command 9-45
Who should read this book viii
Width, memory access 5-35
Window menu in AXD 5-99

## Symbols

! armsd command 9-9
# indicator 6-4, 8-14, 9-7
$ indicator 9-7
$clock internal variable 5-70
$image_cache_enable internal variable 5-70
$rdi_log internal variable 5-69
$statistics internal variable 5-69
$target_fpu internal variable 5-8, 5-70, 8-9, 9-28
+ indicator 6-4
..... symbol 5-45
@ 4-12
@ indicator 4-10, 6-4, 9-7
^ indicator 8-13, 9-7
| armsd command 9-9
| indicator 6-4

 Copyright © 1999-2001 ARM Limited. All rights reserved. 