ARM® Developer Suite

Version 1.1

Debuggers Guide

ARM

ARM DUI 0066C

ARM Developer Suite v1.1
Debuggers Guide

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
October 1999	A	Release 1.0
March 2000	B	Release 1.0.1
November 2000	C	Release 1.1

Proprietary Notice

Contents
Debuggers Guide

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.

About this book

This book has three parts which describe all 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 Parts B and C, but you might have to refer to the Glossary and Index at the end of the book.
- Part B describes the *ARM Debugger for Windows* (ADW) and the *ARM Debugger for UNIX* (ADU). These earlier ARM debuggers continue to be fully supported.
- Part C describes the *ARM Symbolic Debugger* (armsd).

Intended audience

This book is written for developers who are using any 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 xi).

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 various ARM debuggers 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.

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 ADW and ADU.

Chapter 7 ***About ADW and ADU***

Chapter 7 introduces the ADW and ADU debuggers.

Chapter 8 ***Getting Started in ADW and ADU***

Chapter 8 explains how to set up and start using ADW and ADU, and describes some necessary debugging concepts.

Chapter 9 ***Working with ADW and ADU***

Chapter 9 provides detailed descriptions of the features of ADW and ADU, and instructions for their use.

Chapter 10 ***Using ADW and ADU with C++***

Chapter 10 describes how ARM C++ affects ADW and ADU.

PART C Part C covers the use of armsd.

Chapter 11 ***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 12 ***Getting Started in armsd***

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

Chapter 13 ***Working with armsd***

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

Appendix A ***AXD and ADW/ADU/armsd Commands***

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

Appendix B ***Coprocessor Registers***

Appendix B describes the various available coprocessor registers.

Glossary An alphabetically arranged glossary defines the special terms used.

Typographical conventions

The following typographical conventions are used in this book:

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

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/Documentation/Index.html` for current information.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

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 Compiler, Linker, and Utilities Guide* (ARM DUI 0067)
- *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\ARMDUI0100D_armarm.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*. 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 Peripheral Specification* (ARM DDI 0062)
- 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`.

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 you comments apply
- a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Part A

AXD

Chapter 1
About AXD

This chapter explains some of the concepts of debugging and the terminology used. It also describes the various 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 1-11.

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*. Other parts of this book cover earlier ARM debuggers that are still fully supported.

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
- allow 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®
- 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 open ARM standard procedural interface between a debugger and the debug agent (see Figure 1-1 on page 1-6). The widest possible adoption of this standard is encouraged.

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).

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.

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*).

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.

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.

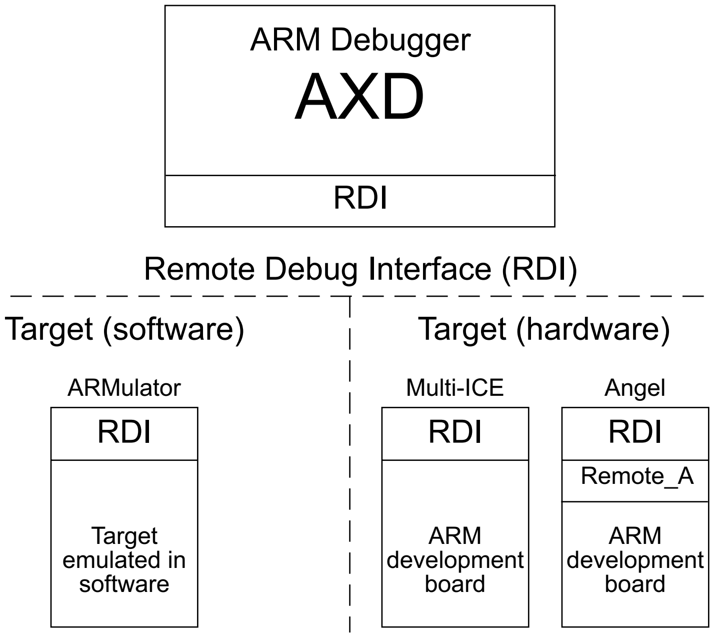


Figure 1-1 Debugger-target interface

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, ADW, ADU, or armsd.

Figure 1-2 shows a typical debugging arrangement of hardware and software:

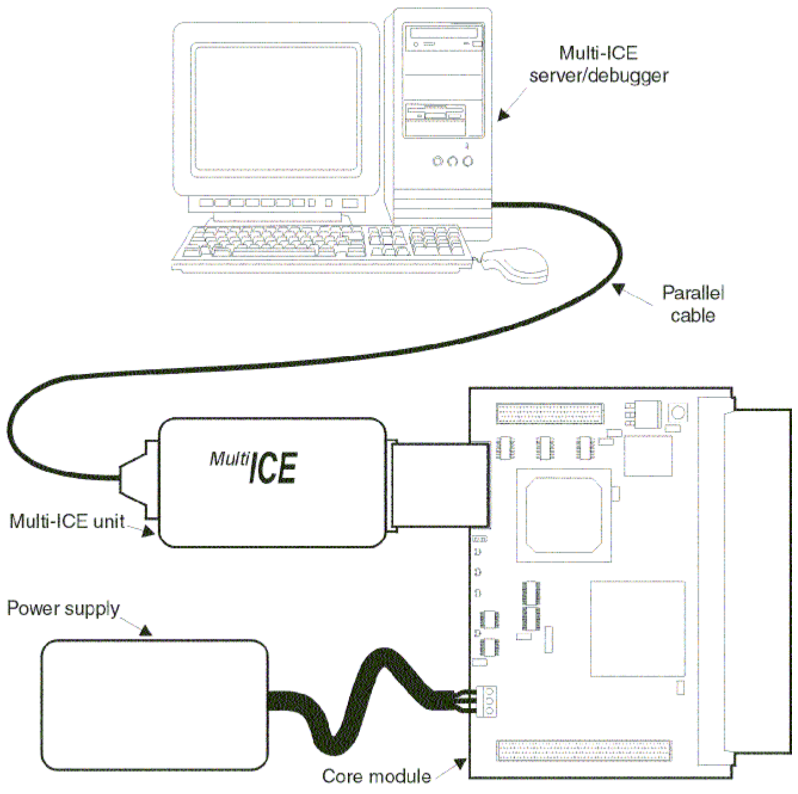


Figure 1-2 A typical debugging set-up

1.3 Debugging systems

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

- *ARMulator* on page 1-8
- *Multi-ICE® and EmbeddedICE®* on page 1-8
- *Angel debug monitor* on page 1-9
- *Gateway* on page 1-9.

See *Configure Target...* on page 5-81 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 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).

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 Gateway

Gateway 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 both to 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.

Gateway handles all aspects of setting up and maintaining the Ethernet connection with the emulator. However, you must provide the correct IP address and specify the core to be debugged using the Gateway configuration dialog (see *Gateway configuration* on page 5-88). It is also your responsibility to ensure that the emulation probe and TIM are suitable for the application core.

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 its authorized agents.

1.4 Availability and compatibility

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

The ADS CD-ROM includes the following ARM debuggers:

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

AXD is the recommended debugger. It provides functionality that is not available in the other debuggers. ADW and ADU will not be supplied in future versions of ADS.

The main improvements in AXD, compared to the earlier ARM debuggers, are:

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

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.

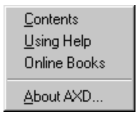


Figure 1-3 Help menu

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 allows you to display online copies of the printed manuals that you received with AXD. This is equivalent to selecting **Start** → **Programs** → **ARM Developer Suite v1.1** → **Online Books**.

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

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

Click on the **?/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.

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.

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:

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 need a license you can obtain it quickly by applying for it by email (preferred) or fax.

You can use some license-managed software with a temporary license, for evaluation purposes. When this is allowed, either a restriction is imposed on functionality or a time limit is placed 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*.

2.2 Starting and closing AXD

This section describes:

- *Starting AXD* on page 2-3
- *AXD arguments* on page 2-3
- *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 v1.1** → **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* on page 2-3). 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* on page 2-3)
- 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 lower case):

```
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. That earlier session is restored to the state it was in when it was saved. If the filename is not fully qualified it is assumed to be in the current directory. If the filename includes spaces, you must enclose it in quotes. Any images in the session file are loaded, and connection to the target is established.

`-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`*.

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.

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* on page 2-6
- *Multi-ICE unit and target board* on page 2-7
- *Angel or EmbeddedICE* on page 2-8
- *Gateway* on page 2-8.

The first time you run AXD, ARMulator is selected by default as the target, with default settings taken from a configuration file. 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

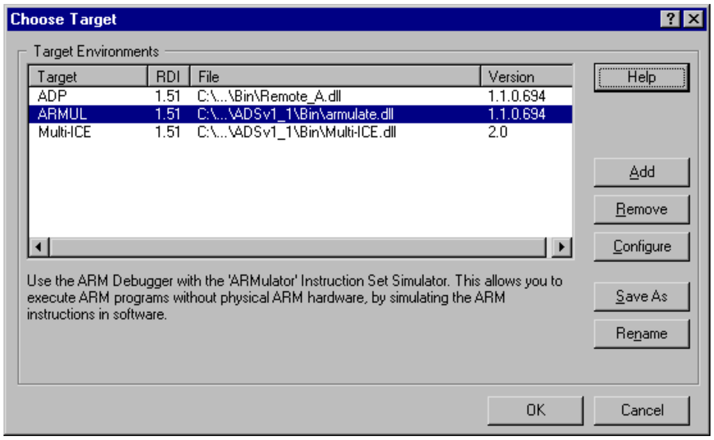
In some of these procedures you use a browse dialog to locate and select a required file such as `armulate.dll`. Some files, including DLLs, are not listed by default. Select **Windows Explorer** → **View** → **Options...** → **Show all files** to display system files.

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. In AXD, 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.



Target	RDI	File	Version
ADP	1.51	C:\...\Bin\Remote_A.dll	1.1.0.694
ARMUL	1.51	C:\...\ADSv1_1\Bin\armulate.dll	1.1.0.694
Multi-ICE	1.51	C:\...\ADSv1_1\Bin\Multi-ICE.dll	2.0

Use the ARM Debugger with the 'ARMulator' Instruction Set Simulator. This allows you to execute ARM programs without physical ARM hardware, by simulating the ARM instructions in software.

Figure 2-1 Selecting a target

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. To examine or change the ARMulator configuration settings, click the **Configure** button. The resulting dialog is described in *Configure Target...* on page 5-81.
4. When you have selected ARMUL as the target, and configured it if necessary, click **OK**. 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 development 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. In AXD, 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 5-81.
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 PID board as the target. An 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 5-65.

2.3.3 Angel or EmbeddedICE

To start an Angel or EmbeddedICE debugging session:

1. Ensure your target board (for example, an ARM development board) is correctly configured and connected to your computer, then switch on its power supply.
2. In AXD, 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.
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. 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 5-81.
5. When you have selected ADP as the target, and configured it if necessary, click **OK**. You can now load an image onto the target and control its execution.

2.3.4 Gateway

To start a Gateway debugging session:

1. In AXD, 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.
2. Click **Add…**. A standard file dialog is displayed.
3. Select `gateway.dll` and click **Open**.
4. Click **Configure…** in the Choose Target screen. The Gateway Remote Configuration panel is displayed, as described in *Configure Target...* on page 5-81.
5. When you have completed your Gateway configuration, click **OK** in the Debugger Configuration screen to restart the debugger with a connection to the probe.

2.4 AXD displays

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

- *Views* on page 2-9
- *Viewing structured data* on page 2-9
- *Multi-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 5-19.
- 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 5-44.

2.4.2 Viewing structured data

In Register, Variable, 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 Register 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 Variable 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.

2.4.3 Multi-document interface

AXD uses the Windows *Multi-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 various 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.

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.

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

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*.

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

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

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

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

Execute Allows you to set or edit breakpoints and watchpoints, and control execution of a program image.

Options Allows 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 Allows you to control how MDI windows and icons are displayed, and to set refresh options.

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

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.

2.6 Tool icons, status bar, keys, and commands

This section introduces:

- *Toolbars* on page 2-14
- *Tooltips* on page 2-14
- *Status bar* on page 2-14
- *Keyboard shortcuts* on page 2-14
- *In-place editing* on page 2-15
- *Command-line interface* on page 2-15.

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* on page 5-5).

You can display or hide the status bar (see *Status Bar* on page 5-93).

2.6.4 Keyboard shortcuts

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

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:

- **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.

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.

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:

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_1` 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 development 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 5-81.

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

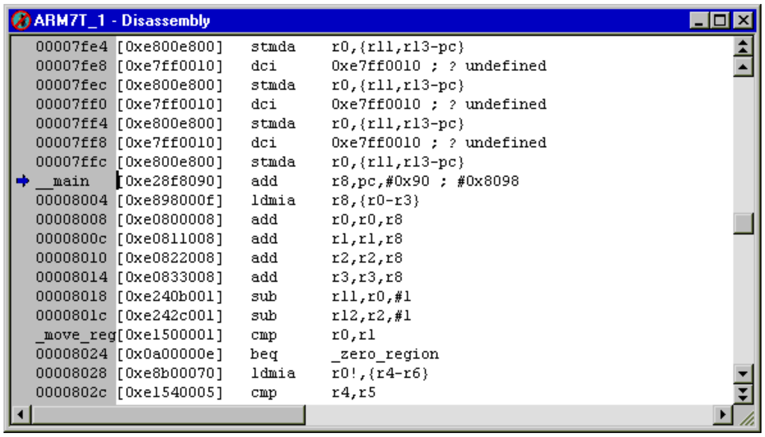
1. Create an executable image by compiling the source code files in the `Dhry` subdirectory and linking the resulting objects with the libraries that they use. If you are running under Windows you can use the CodeWarrior IDE project file `dhry.mcp` supplied. This organizes your work into projects and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



```
  00007fe4 [0xe800e800]   stmda    r0,{r11,r13-pc}
  00007fe8 [0xe7ff0010]   dci      0xe7ff0010 ; ? undefined
  00007fec [0xe800e800]   stmda    r0,{r11,r13-pc}
  00007ff0 [0xe7ff0010]   dci      0xe7ff0010 ; ? undefined
  00007ff4 [0xe800e800]   stmda    r0,{r11,r13-pc}
  00007ff8 [0xe7ff0010]   dci      0xe7ff0010 ; ? undefined
  00007ffc [0xe800e800]   stmda    r0,{r11,r13-pc}
➜ __main   [0xe28f8090]   add      r8,pc,#0x90 ; #0x8098
  00008004 [0xe898000f]   ldmia    r8,{r0-r3}
  00008008 [0xe0800008]   add      r0,r0,r8
  0000800c [0xe0811008]   add      r1,r1,r8
  00008010 [0xe0822008]   add      r2,r2,r8
  00008014 [0xe0833008]   add      r3,r3,r8
  00008018 [0xe240b001]   sub      r11,r0,#1
  0000801c [0xe242c001]   sub      r12,r2,#1
  _move_reg[0xe1500001]   cmp      r0,r1
  00008024 [0x0a00000e]   beq      _zero_region
  00008028 [0xe8b00070]   ldmia    r0!,{r4-r6}
  0000802c [0xe1540005]   cmp      r4,r5
```

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 indicates 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 indicates 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.

3.2 Setting a breakpoint

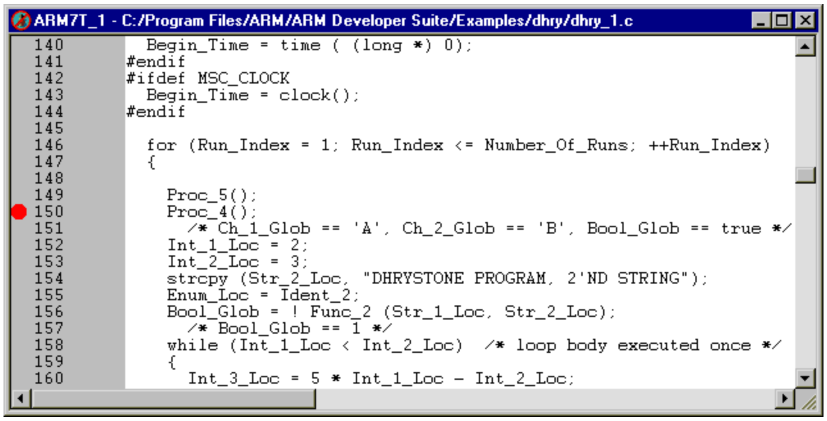
This example runs the same program again, this time with a breakpoint that stops execution a few times. You can examine values when execution stops.

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 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 indicates that you have set a second breakpoint, as shown in Figure 3-2.



ARM7T_1 - C:/Program Files/ARM/ARM Developer Suite/Examples/dhry/dhry_1.c

```
  140      Begin_Time = time ( (long *) 0);
  141    #endif
  142    #ifdef MSC_CLOCK
  143      Begin_Time = clock();
  144    #endif
  145
  146      for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
  147      {
  148
  149        Proc_5();
● 150        Proc_4();
  151          /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
  152        Int_1_Loc = 2;
  153        Int_2_Loc = 3;
  154        strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
  155        Enum_Loc = Ident_2;
  156        Bool_Glob = ! Func_2 (Str_1_Loc, Str_2_Loc);
  157          /* Bool_Glob == 1 */
  158        while (Int_1_Loc < Int_2_Loc)  /* loop body executed once */
  159        {
  160          Int_3_Loc = 5 * Int_1_Loc - Int_2_Loc;
```

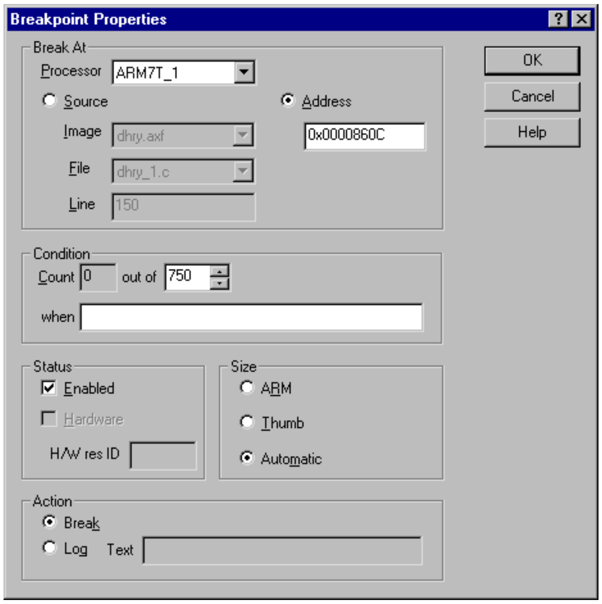
Figure 3-2 Breakpoint set inside loop

5. To edit the details of the new breakpoint, select **Breakpoints** from the **System Views** menu. 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 Conditions box, 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.

   Click **OK**.


Breakpoint Properties

Break At
Processor: ARM7T_1
Source [✓ Address]
Image: dhry.axf
File: dhry_1.c
Line: 150
Address: 0x0000860C

Condition
Count: 0 out of 750
when:

Status
[✓ Enabled]
Hardware
H/W res ID:

Size
ARM
Thumb
[✓ Automatic]

Action
[✓ Break]
Log Text:

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.

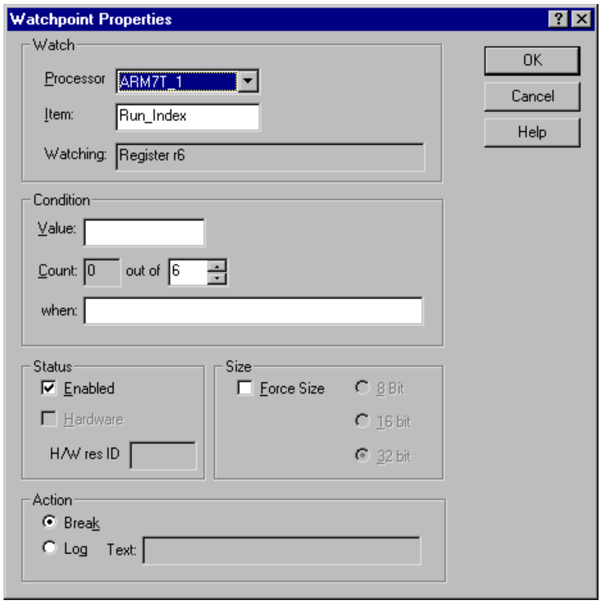
9. Press F5 repeatedly until the value of `Run_Index` reaches the highest multiple of 750 before exceeding your specified number of runs, then once more to allow the program to complete execution. (This time the Dhrystone test results are meaningless, because of the interruptions to the timing measurements,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

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. 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. When you are prompted for the number of runs to execute, enter 770. 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.


Watchpoint Properties

Watch
Processor: ARM7T_1
Item: Run_Index
Watching: Register r6

Condition
Value:
Count: 0 out of 6
when:

Status
[✓ Enabled]
Hardware
H/W res ID

Size
Force Size
8 Bit
16 bit
[✓ 32 bit]

Action
[✓ Break]
Log Text:

Figure 3-4 Setting a watchpoint

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

Set the out of field in the Conditions box 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. If the Variables processor view is not already displayed, select **Variables** from the **Processor Views** menu. 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.

3.4 Examining the contents of variables

Two methods of examining the contents of variables are described:

- *Contents of variables* on page 3-8.

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

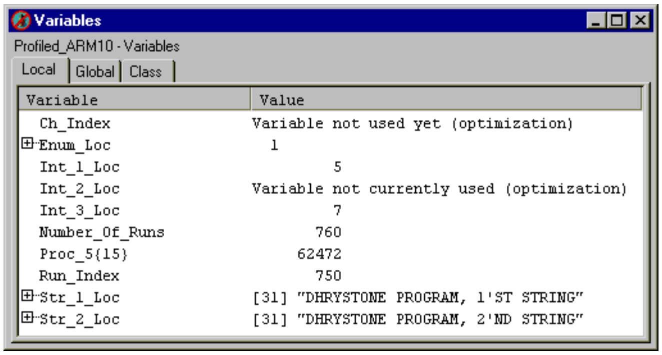
  This method shows the addresses of the variables as well as their contents.

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. When you are prompted for the number of runs to execute, enter 760. 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. If the Variables processor view is not already displayed, select **Variables** from the **Processor Views** menu. Reposition or resize the window if necessary. On the Local tabbed page 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.



Variable	Value
Ch_Index	Variable not used yet (optimization)
⊞Enum_Loc	1
Int_1_Loc	5
Int_2_Loc	Variable not currently used (optimization)
Int_3_Loc	7
Number_Of_Runs	760
Proc_5{15}	62472
Run_Index	750
⊞Str_1_Loc	[31] "DHRYSTONE PROGRAM, 1'ST STRING"
⊞Str_2_Loc	[31] "DHRYSTONE PROGRAM, 2'ND STRING"

Figure 3-5 Examining the contents of variables

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 4-15.

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

3.4.2 Addresses and contents of variables

An alternative method of examining a variable is to use a Watch processor view. This allows 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. When you are prompted for the number of runs to execute, enter 760. 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 tabbed page only.

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 on page 3-10.

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

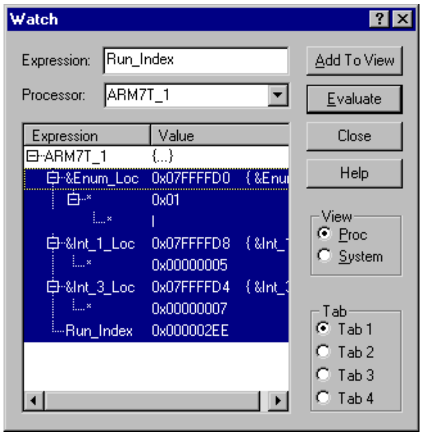
You can also add a variable to the Watch view by selecting it in the source view and using the **Add Watch** pop-up menu command.

---

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.



Expression: Run_Index

Processor: ARM7T_1

View: [✓ Proc] System

Tab: [✓ Tab 1] Tab 2 Tab 3 Tab 4

```
Expression       Value
⊟ARM7T_1         {...}
  ⊟&Enum_Loc     0x07FFFFD0   { &Enu
    ⊟*           0x01
      *          |
  ⊟&Int_1_Loc    0x07FFFFD8   { &Int_
    *            0x00000005
  ⊟&Int_3_Loc    0x07FFFFD4   { &Int_
    *            0x00000007
  Run_Index      0x000002EE
```

Figure 3-6 Specifying variables to watch

Enter, in a similar way:

`&Int_1_Loc`

`&Int_3_Loc`

`Run_Index`

Expand these lines also.

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.

   Point 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 4-15.

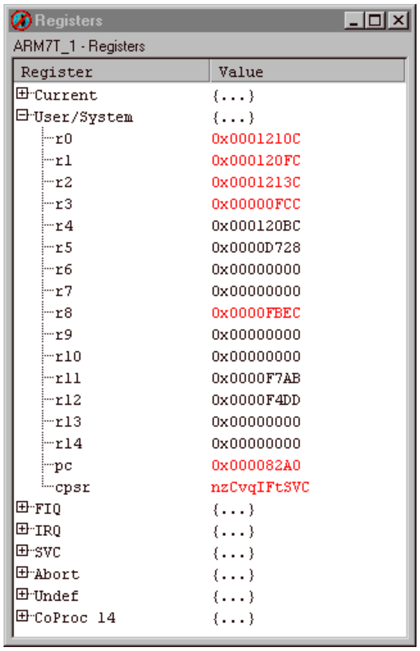
13. Press F5 to allow the program to complete its execution, then close down the Watch processor view.

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.



```
Register          Value
⊞ Current         {...}
⊟ User/System     {...}
    r0            0x0001210C
    r1            0x000120FC
    r2            0x0001213C
    r3            0x00000FCC
    r4            0x000120BC
    r5            0x0000D728
    r6            0x00000000
    r7            0x00000000
    r8            0x0000FBEC
    r9            0x00000000
    r10           0x00000000
    r11           0x0000F7AB
    r12           0x0000F4DD
    r13           0x00000000
    r14           0x00000000
    pc            0x000082A0
    cpsr          nzCvqIFtSVC
⊞ FIQ             {...}
⊞ IRQ             {...}
⊞ SVC             {...}
⊞ Abort           {...}
⊞ Undef           {...}
⊞ CoProc 14       {...}
```

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.

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 4-15.

   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.

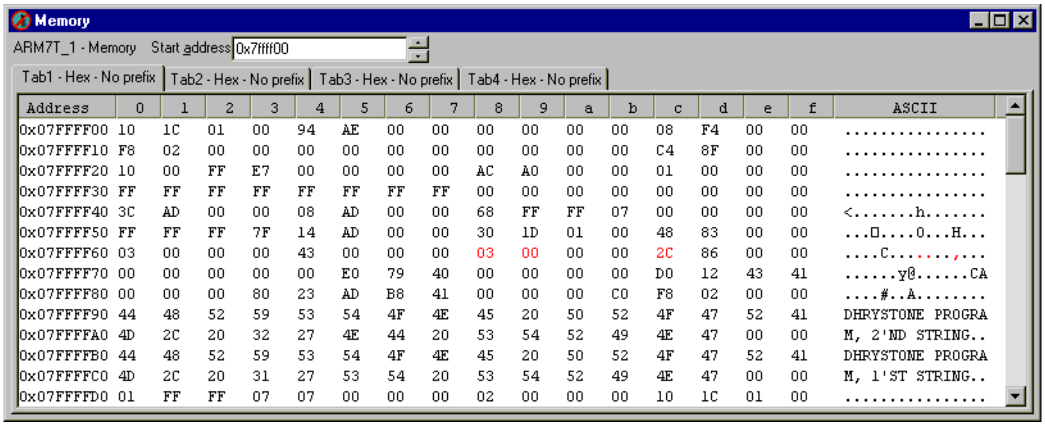
   You can also select **Add Register** from the pop-up menu of the Registers system view. This allows you to select registers from any processor to add to those being displayed in the Registers system view.

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

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. When you are prompted for the number of runs to execute, enter 760. 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.



```
0x7ffff00

Address     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f     ASCII
0x07FFFF00 10 1C 01 00 94 AE 00 00 00 00 00 00 08 F4 00 00  ................
0x07FFFF10 F8 02 00 00 00 00 00 00 00 00 00 00 C4 8F 00 00  ................
0x07FFFF20 10 00 FF E7 00 00 00 00 AC A0 00 00 01 00 00 00  ................
0x07FFFF30 FF FF FF FF FF FF FF FF 00 00 00 00 00 00 00 00  ................
0x07FFFF40 3C AD 00 00 08 AD 00 00 68 FF FF 07 00 00 00 00  <.......h.......
0x07FFFF50 FF FF FF 7F 14 AD 00 00 30 1D 01 00 48 83 00 00  ...□....0...H...
0x07FFFF60 03 00 00 00 43 00 00 00 03 00 00 00 2C 86 00 00  ....C.......,...
0x07FFFF70 00 00 00 00 00 E0 79 40 00 00 00 00 D0 12 43 41  ......y@......CA
0x07FFFF80 00 00 00 80 23 AD B8 41 00 00 00 C0 F8 02 00 00  ....#..A........
0x07FFFF90 44 48 52 59 53 54 4F 4E 45 20 50 52 4F 47 52 41  DHRYSTONE PROGRA
0x07FFFFA0 4D 2C 20 32 27 4E 44 20 53 54 52 49 4E 47 00 00  M, 2'ND STRING..
0x07FFFFB0 44 48 52 59 53 54 4F 4E 45 20 50 52 4F 47 52 41  DHRYSTONE PROGRA
0x07FFFFC0 4D 2C 20 31 27 53 54 20 53 54 52 49 4E 47 00 00  M, 1'ST STRING..
0x07FFFFD0 01 FF FF 07 07 00 00 00 02 00 00 00 10 1C 01 00  ................
```

Figure 3-8 Examining contents of memory

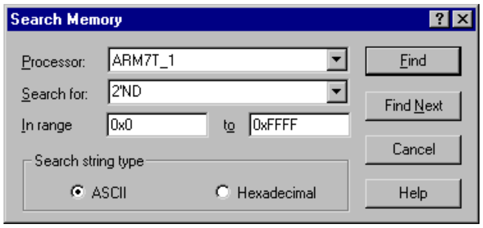
Addresses and contents of variables on page 3-9 shows that addresses of interest are in the region of `0x07FFFFD0`, so set the Start address value to, say, `0x07FFFF00`.

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.
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*.

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, enter `2'ND` in the Search for field, set the In range and to addresses to `0x0` and `0xFFFF`, select ASCII for the Search string type, and click the **Find** button, as shown in Figure 3-9. Click **Cancel**.


Search Memory

Processor: ARM7T_1

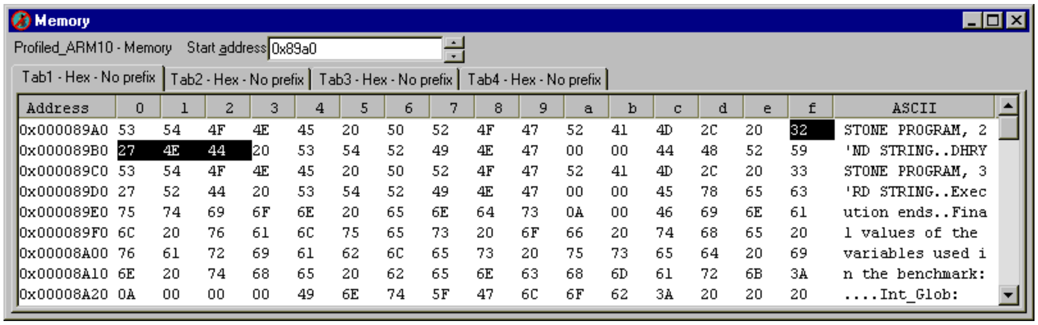
Search for: 2'ND

In range 0x0 to 0xFFFF

Search string type: [✓ ASCII] Hexadecimal

Figure 3-9 Searching for a string in memory

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.


Memory

Profiled_ARM10 - Memory Start address 0x89a0

```
Address     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   ASCII
0x000089A0 53 54 4F 4E 45 20 50 52 4F 47 52 41 4D 2C 20 32   STONE PROGRAM, 2
0x000089B0 27 4E 44 20 53 54 52 49 4E 47 00 00 44 48 52 59   'ND STRING..DHRY
0x000089C0 53 54 4F 4E 45 20 50 52 4F 47 52 41 4D 2C 20 33   STONE PROGRAM, 3
0x000089D0 27 52 44 20 53 54 52 49 4E 47 00 00 45 78 65 63   'RD STRING..Exec
0x000089E0 75 74 69 6F 6E 20 65 6E 64 73 0A 00 46 69 6E 61   ution ends..Fina
0x000089F0 6C 20 76 61 6C 75 65 73 20 6F 66 20 74 68 65 20   l values of the
0x00008A00 76 61 72 69 61 62 6C 65 73 20 75 73 65 64 20 69   variables used i
0x00008A10 6E 20 74 68 65 20 62 65 6E 63 68 6D 61 72 6B 3A   n the benchmark:
0x00008A20 0A 00 00 00 49 6E 74 5F 47 6C 6F 62 3A 20 20 20   ....Int_Glob:
```

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`.

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

   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.

5. 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.

3.8 Creating a revised version of the program

In *Locating and changing values and verifying changes* on page 3-15, 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*.

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:

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:

- *Breakpoint* on page 4-2
- *Watchpoint* on page 4-2
- *Stepping through a program* on page 4-3.

Detailed descriptions of how to use these facilities are given in *Execute menu* on page 5-70, and in the online help.

4.1.1 Breakpoint

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 5-55, and toggling (switching on and off) in *Toggle Breakpoint* on page 5-71. You can also set breakpoints in some processor views (see *Source... processor view* on page 5-41, *Disassembly processor view* on page 5-39, and *Memory processor view* on page 5-31).

4.1.2 Watchpoint

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.

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 5-58.

4.1.3 Stepping through a program

Once 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 5-70.

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.

4.2 Expressions

This section describes:

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

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:

  ```
  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
  ```

- Though it is possible to call member functions in the form `Class::Member(...)`, this gives undefined results.

- **`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`

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* on page 4-6
- *Source files* on page 4-6
- *Disassembled code* on page 4-7
- *Registers* on page 4-7
- *Watch* on page 4-7
- *Variables* on page 4-8
- *Memory* on page 4-9
- *Remote debug information* on page 4-9
- *High-level and low-level symbols* on page 4-10
- *Debugger internals* on page 4-10
- *Backtrace* on page 4-10
- *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 5-45.

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.

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 5-41.

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 5-39.

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 5-20. To select registers from various Registers processor views to display together in a single Registers system view, see *Registers system view* on page 5-51.

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.
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 allows 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 5-24 and *Watch system view* on page 5-53, 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 allows you to update the value.

For further details, see *Variables processor view* on page 5-27.

4.3.7 Memory

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

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 allows you to update the value.

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

Locate to memory

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

- Registers
- Variables
- Watch
- Backtrace
- Memory
- Low Level Symbols.

In any of these views, if you select a data item that can be interpreted as a memory address, then select **Locate To Memory** 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 pop-up menu* on page 5-25.

4.3.8 Remote debug information

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

For further information, see *Output system view* on page 5-60.

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.

In a regular expression, precede the symbol with `@` to indicate a low-level symbol.

For further information, see *Low Level Symbols processor view* on page 5-34.

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 screens:

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

For further information, see *Debugger Internals system view* on page 5-65.

4.3.11 Backtrace

A call stack is maintained for each processor in the target, and the Backtrace processor view allows 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.

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 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 5-36.

4.3.13 Semihosting

The Console view allows 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 5-38.

4.4 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 5-74.

You can also choose to restart any previous session that was saved in a session file (see *Load Session...* on page 5-13 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 extention `.ses`. See *Save Session...* on page 5-14. You can create any number of session files. To restart an earlier session, see *Load Session...* on page 5-13.

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** checkbox determines what happens. This checkbox is on the General tab of the Configure Interface dialog (see *Configure Interface...* on page 5-74). If checked, the previous session is restarted. If unchecked, system default settings are used and no earlier session is restarted.

4.5 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 of 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 5-74 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.

If you select **Properties...** from the Memory or Backtrace processor view, you can select or deselect an **Automatic refresh** checkbox. 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.

4.6 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:

- register views
- watch views
- variable views
- debugger internal 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.

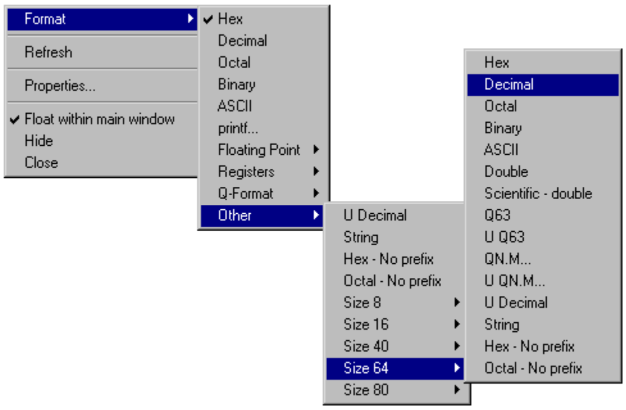


Figure 4-1 Some format submenus

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.

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 4-16
- *Decimal* on page 4-17
- *Octal* on page 4-17
- *Binary* on page 4-17
- *ASCII* on page 4-18
- *Printf...* on page 4-18
- *Floating point* on page 4-18
- *Registers* on page 4-19
- *Q-format* on page 4-22
- *Other* on page 4-23.

4.6.1 Hex

A 2-character prefix of a figure `0` and an upper case 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
```

4.6.2 Decimal

A value in decimal format has no prefix. Digits (`0-9`) 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.

4.6.3 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`.

4.6.4 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...* on page 4-18).

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.

4.6.5 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.

4.6.6 Printf...

This displays a dialog allowing you to use standard C formats to specify the format to be used for displaying the currently selected data. Examples are:

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

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`.

4.6.7 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.

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

4.6.8 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 5-20 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`).

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 `xuOZI_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 `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.

4.6.9 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.

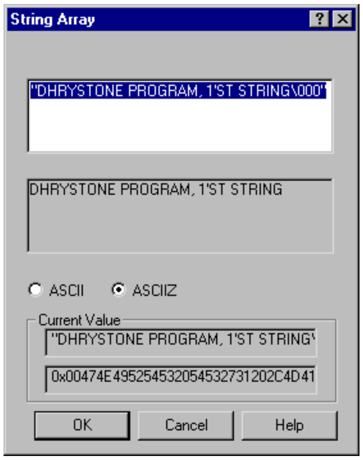
4.6.10 Other

This submenu includes all the remaining available formats that do not appear on other other submenus. It also allows 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 4-17.

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 Array dialog is displayed, as shown in Figure 4-2.



"DHRYSTONE PROGRAM, 1'ST STRING\000"

DHRYSTONE PROGRAM, 1'ST STRING

( ) ASCII [✓ ASCIIZ]

Current Value

"DHRYSTONE PROGRAM, 1'ST STRING\

0x00474E495254532054532731202C4D41

Figure 4-2 String Array 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.

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.

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 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.

4.7 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 and Angel, but not by EmbeddedICE or Multi-ICE.

To collect profiling information when executing an image, you must make certain settings when you load the image (see *Load Image...* on page 5-7) or before reloading the image (see Figure 5-59 on page 5-48).

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. When the image terminates, select **Options** → **Profiling** → **Write to File**.
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 Compiler, Linker, and Utilities 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.
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.

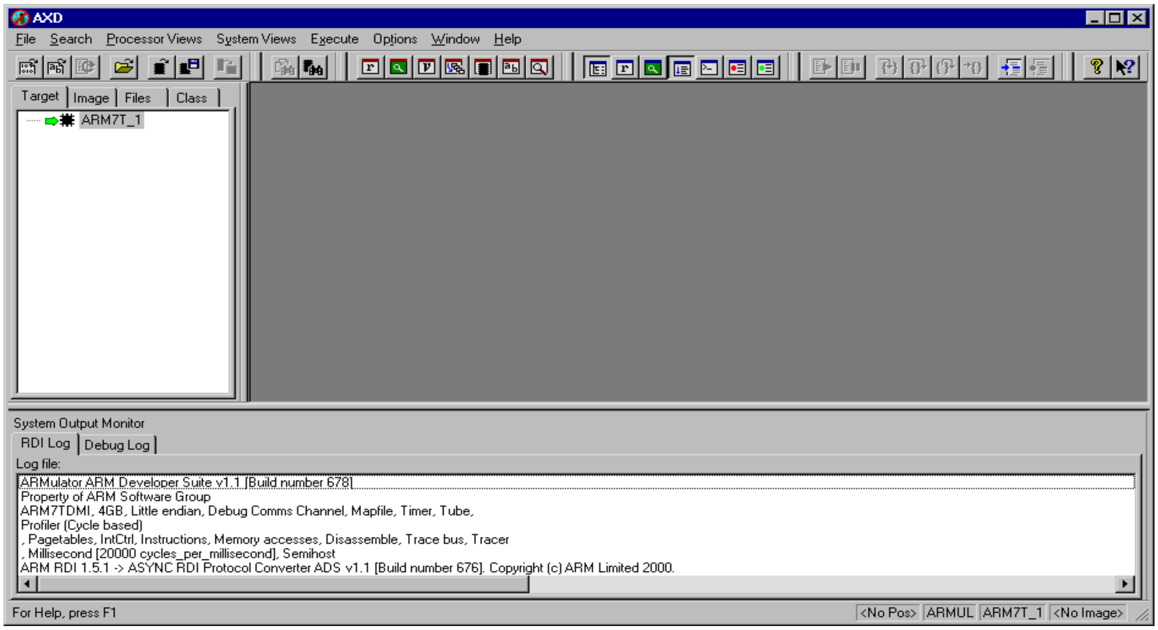
Chapter 5
AXD Desktop

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:

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 4-12).


AXD
File Search Processor Views System Views Execute Options Window Help
Target Image Files Class
ARM7T_1
System Output Monitor
RDI Log Debug Log
Log file:
For Help, press F1
<No Pos> ARMUL ARM7T_1 <No Image>

```
ARMulator ARM Developer Suite v1.1 [Build number 678]
Property of ARM Software Group
ARM7TDMI, 4GB, Little endian, Debug Comms Channel, Mapfile, Timer, Tube,
Profiler (Cycle based)
, Pagetables, IntCtrl, Instructions, Memory accesses, Disassemble, Trace bus, Tracer
, Millisecond [20000 cycles_per_millisecond], Semihost
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter ADS v1.1 [Build number 676]. Copyright (c) ARM Limited 2000.
```

Figure 5-1 AXD opening screen

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

- *Menus* on page 5-2
- *Toolbars* on page 5-3
- *Status bar* 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 5-19 and *System Views menu* on page 5-44 include details of pop-up menus.

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 5-74). 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.

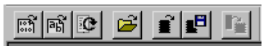


Figure 5-2 File toolbar

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.

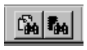


Figure 5-3 Search toolbar

These tools are described as menu items in *Search menu* on page 5-17.

Processor Views toolbar

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

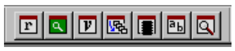


Figure 5-4 Processor Views toolbar

These tools are described as menu items in *Processor Views menu* on page 5-19.

System Views toolbar

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

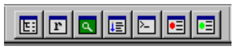


Figure 5-5 System Views toolbar

These tools are described as menu items in *System Views menu* on page 5-44.

Execute toolbar

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

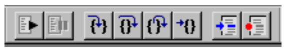


Figure 5-6 Execute toolbar

These tools are described as menu items in *Execute menu* on page 5-70.

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 5-97.

5.1.3 Status bar

If you choose to display the status bar (see page 5-93) it appears at the bottom of the AXD screen, as shown in Figure 5-8.

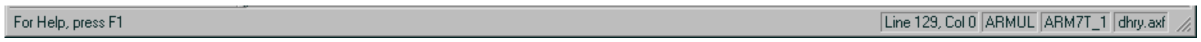


Figure 5-8 Status bar

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.

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.

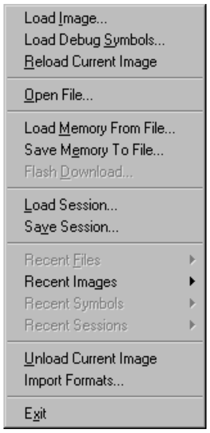


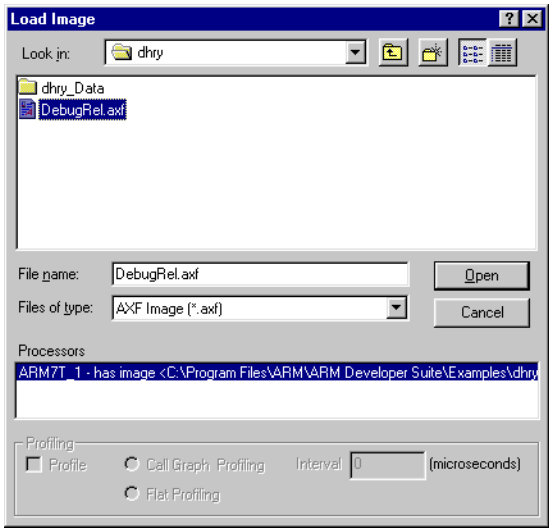
Figure 5-9 File menu

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 5-8
- *Open File...* on page 5-9
- *Load Memory From File...* on page 5-10
- *Save Memory To File...* on page 5-11
- *Flash Download...* on page 5-12
- *Load Session...* on page 5-13
- *Save Session...* on page 5-14
- *Recent Files* on page 5-14
- *Recent Images* on page 5-15
- *Recent Symbols* on page 5-15
- *Recent Sessions* on page 5-15
- *Unload Current Image* on page 5-16
- *Import Formats...* on page 5-16
- *Exit* on page 5-16.

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.



File name: DebugRel.axf

Files of type: AXF Image (*.axf)

Processors

ARM7T_1 - has image <C:\Program Files\ARM\ARM Developer Suite\Examples\dhry

Interval 0

Figure 5-10 Selecting an image file to load

Navigate to the directory in which 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 allows 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** box and set the other profiling details in this dialog before loading the image.

Flat profiling accumulates limited information without altering the image. Call-graph profiling accumulates more detailed information but has to add extra code to the 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 4-25).

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 5-65.

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.

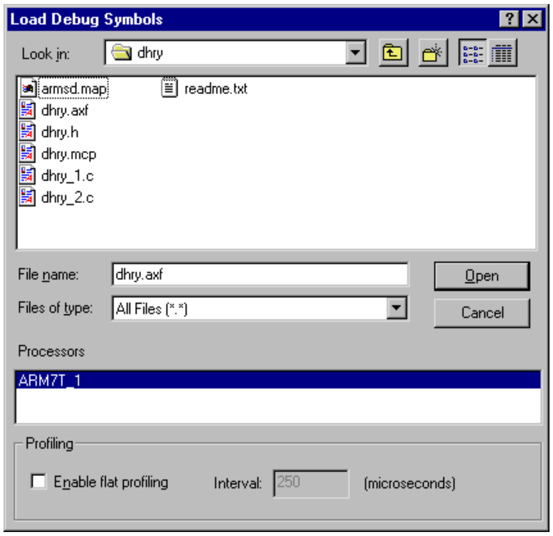


Figure 5-11 Load Debug Symbols dialog

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.

Leave the **Enable flat profiling** check box unchecked if you do not intend to collect any profiling information from this image. The ability to specify flat profiling from this dialog allows flat profiling of images running under RealMonitor. 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 4-25).

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 5-59 on page 5-48).

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.

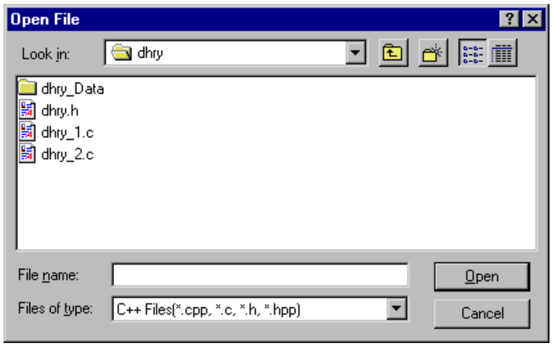


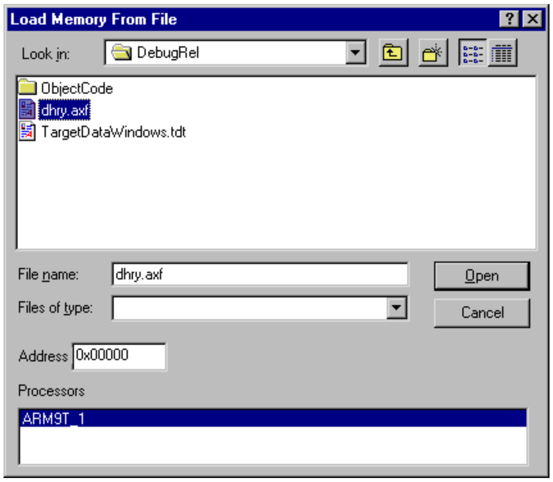
Figure 5-12 Selecting a source file to open

Navigate to the directory in which 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.



File name: dhry.axf
Files of type:
Address 0x00000
Processors
ARM9T_1

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.

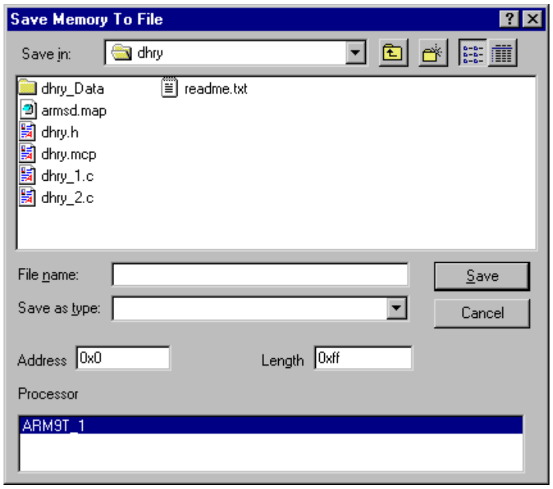
5.2.6 Save Memory To File...

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. This dialog allows 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.

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

Select the directory in which 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.


Save Memory To File

Save in: dhry

dhry_Data
armsd.map
dhry.h
dhry.mcp
dhry_1.c
dhry_2.c
readme.txt

File name:
Save as type:
Address 0x0
Length 0xff
Processor
ARM9T_1

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.

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.

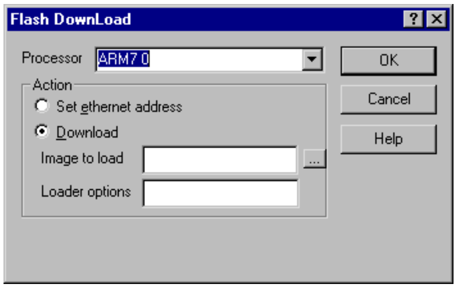


Figure 5-15 Flash Download dialog

2. In the Processor field, select the processor that has the Flash memory into which you want to load an image.
3. In the Action box 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 box 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 the *ADS Compiler, Linker, and Utilities Guide* 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.

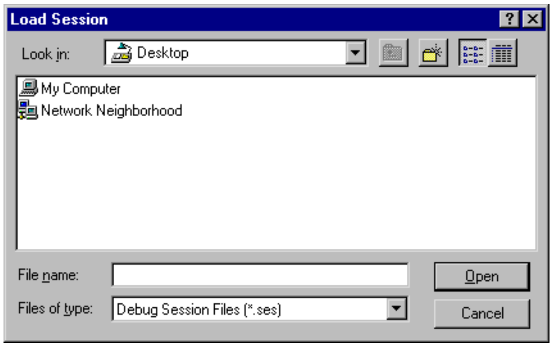


Figure 5-16 Load Session dialog

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 5-15.

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.

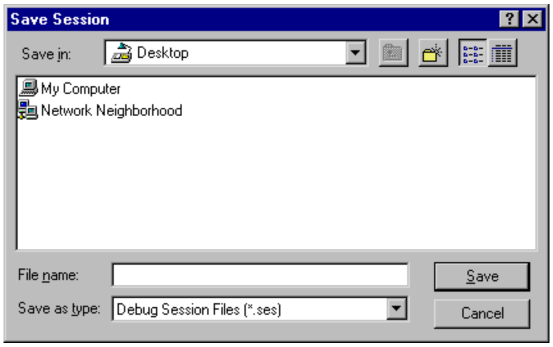


Figure 5-17 Save Session dialog

Change to the directory in which 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.

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 allows 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.

5.2.16 Exit

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

5.3 Search menu

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

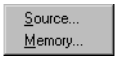


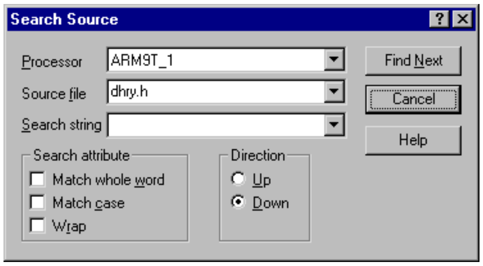
Figure 5-18 Search menu

The Search menu items are described under the following headings:

- *Source...* on page 5-17
- *Memory...* on page 5-18.

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, allows you to specify the target character string, and the file to be searched.


Search Source

Processor: ARM9T_1
Source file: dhry.h
Search string:

Search attribute
- [ ] Match whole word
- [ ] Match case
- [ ] Wrap

Direction
- ○ Up
- ● Down

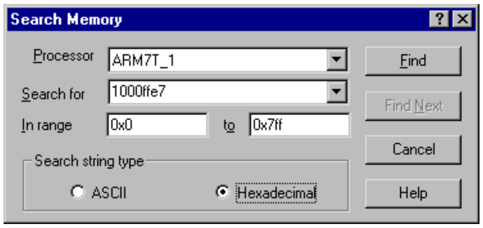
Figure 5-19 Searching for a string in a source file

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 allows 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, allows you to specify what to search for and where to search.


Search Memory

Processor: ARM7T_1
Search for: 1000ffe7
In range: 0x0 to 0x7ff
Search string type: ASCII [✓ Hexadecimal]

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, using hexadecimal notation.

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 allows 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.

5.4 Processor Views menu

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

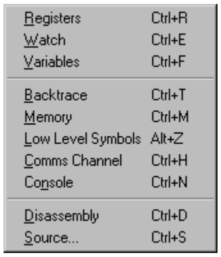


Figure 5-21 Processor Views menu

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 5-45). 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 5-44).

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.

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

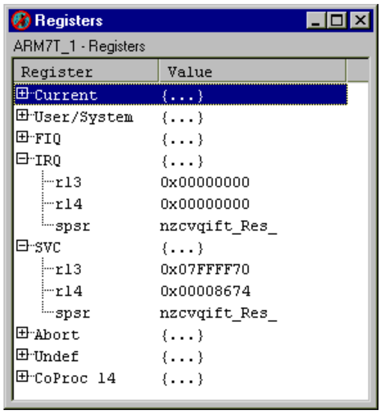
- *Registers processor view* on page 5-20
- *Watch processor view* on page 5-24
- *Variables processor view* on page 5-27
- *Backtrace processor view* on page 5-29
- *Memory processor view* on page 5-31
- *Low Level Symbols processor view* on page 5-34
- *Comms Channel processor view* on page 5-36
- *Console processor view* on page 5-38
- *Disassembly processor view* on page 5-39
- *Source... processor view* on page 5-41.

5.4.1 Registers processor view

The Registers processor view allows you to examine the value of any of the registers in a specific processor. It also allows 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.

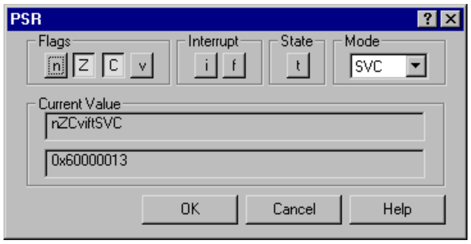


```
Register        Value
⊞Current        {...}
⊞User/System    {...}
⊞FIQ            {...}
⊟IRQ            {...}
   r13          0x00000000
   r14          0x00000000
   spsr         nzcvqift_Res_
⊟SVC            {...}
   r13          0x07FFFF70
   r14          0x00008674
   spsr         nzcvqift_Res_
⊞Abort          {...}
⊞Undef          {...}
⊞CoProc 14      {...}
```

Figure 5-22 Registers processor view

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.

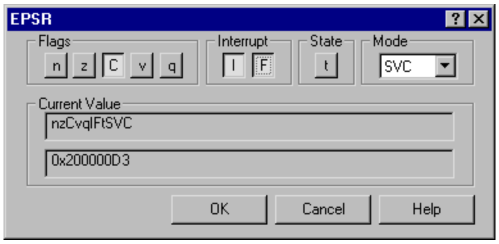
Double-click on the value of any register that you want to change. In-place editing is invoked whenever possible, otherwise a dialog is displayed. Double-clicking on the value of a *Program Status Register* (PSR), for example, displays the dialog shown in Figure 5-23. For more information about data display formats and data entry formats, see *Data Formatting* on page 4-15.



Current Value
nZCviftSVC
0x60000013

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 if you edit the value of that register the dialog in Figure 5-24 is displayed.



Current Value
nzCvqIFtSVC
0x200000D3

Figure 5-24 Enhanced 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.

To add one of the registers displayed in a Registers processor view to the Registers system view (see *Registers system view* on page 5-51), 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* on page 5-22).

Registers processor view pop-up menu

To display the Registers pop-up menu, shown in Figure 5-25, right-click within the Registers processor view.

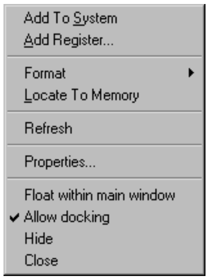


Figure 5-25 Registers processor view pop-up menu

The **Add To System** and **Format** 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.

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 5-26.

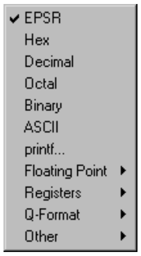


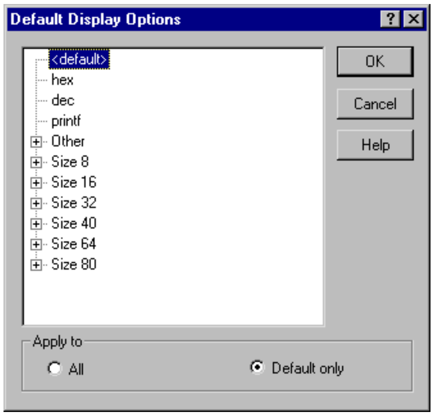
Figure 5-26 Formats available for displaying registers

Refer to *Data Formatting* on page 4-15 for details of the formats available.

The **Locate To Memory** menu item functions as described in *Watch processor view pop-up menu* on page 5-25.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

Select **Properties...** to display the Default Display Options dialog shown in Figure 5-27 on page 5-23.



( ) All [✓ Default only]

Figure 5-27 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 allows 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.

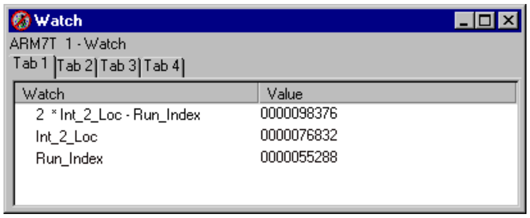
A Watch processor view is initially empty. You choose what is to be listed and have its value shown. One way to add a line to a Watch processor view is to right-click on an item in a Variables processor view and select **Add to Processor Watch** from the resulting pop-up menu.

Another way to add a line is to select **Add Watch** from the pop-up menu (see Figure 5-29 on page 5-25). 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 4-13.)

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 need. 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 5-28. For more information about data display formats and data entry formats, see *Data Formatting* on page 4-15.



Watch	Value
2 * Int_2_Loc - Run_Index	0000098376
Int_2_Loc	0000076832
Run_Index	0000055288

Figure 5-28 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 5-29, right-click within the Watch processor view.

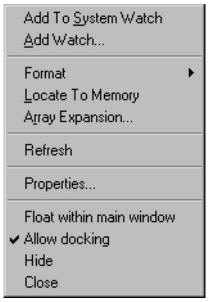
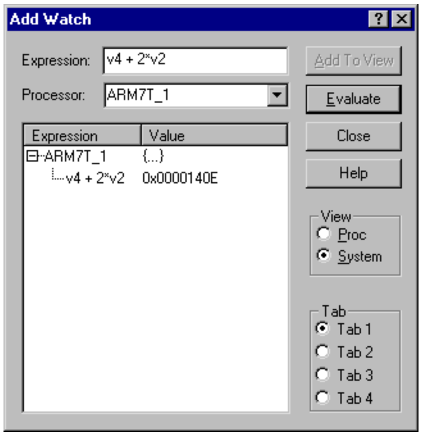


Figure 5-29 Watch processor view pop-up menu

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 5-53).

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 5-30.



Expression: v4 + 2*v2
Processor: ARM7T_1

Expression	Value
ARM7T_1	{...}
v4 + 2*v2	0x0000140E

View: [✓ System]
Tab: [✓ Tab 1]

Figure 5-30 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 5-53), and on which tabbed page it must appear.

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.

Select **Locate To Memory** after selecting a data item that holds a value that can be interpreted as a memory address. The 32 least significant bits only of the selected item are used. 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 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.

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 allows 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.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

Select **Properties...** to display the Default Display Options dialog shown in Figure 5-27 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.

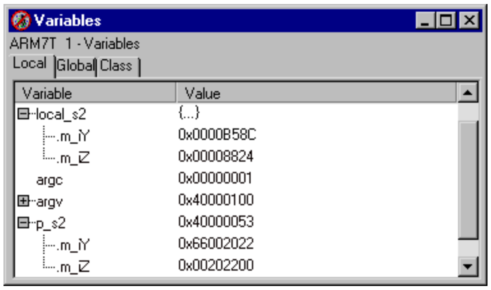
5.4.3 Variables processor view

The Variables processor view allows you to examine and change the value of any of the listed variables.

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 5-31, with its Local tab selected.



Variable	Value
⊟ local_s2	{...}
m_iY	0x0000B58C
m_iZ	0x00008824
argc	0x00000001
⊞ argv	0x40000100
⊟ p_s2	0x40000053
m_iY	0x66002022
m_iZ	0x00202200

Figure 5-31 Variables processor view

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.

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 4-15.

Variables processor view pop-up menu

To display the Variables pop-up menu, shown in Figure 5-32, right-click within the Variables processor view.

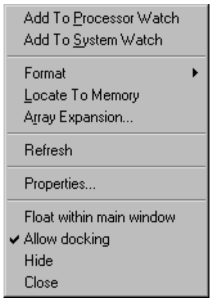


Figure 5-32 Variables processor view pop-up menu

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.

Select **Add To Processor Watch** to add the selected variable to a Watch processor view (see *Watch processor view* on page 5-24).

Select **Add To System Watch** to add the selected variable to a Watch system view (see *Watch system view* on page 5-53).

The **Locate To Memory** and **Array Expansion...** menu items function as described in *Watch processor view pop-up menu* on page 5-25.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

Select **Properties...** to display the Default Display Options dialog shown in Figure 5-27 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).

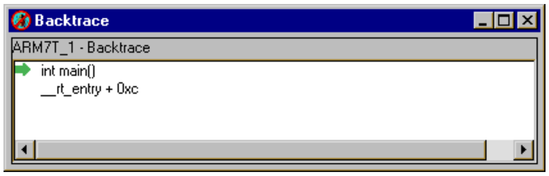
If you close a Variables processor view then later select it again, it is displayed as though you are selecting it for the first time.

5.4.4 Backtrace processor view

The Backtrace processor view allows 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 5-33.



ARM7T_1 - Backtrace

```
int main()
__rt_entry + 0xc
```

Figure 5-33 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.

A line containing three dots indicates a discontinuity in the stack. 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.

It is possible for an application program to overwrite and damage the call stack. A line containing `...//...` indicates that an inconsistency has been detected and the call stack is considered broken.

Backtrace processor view pop-up menu

To display the Backtrace pop-up menu, shown in Figure 5-34, right-click within the Backtrace processor view.

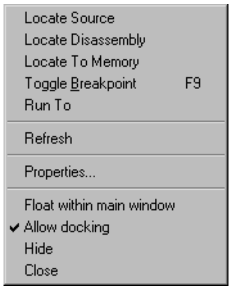


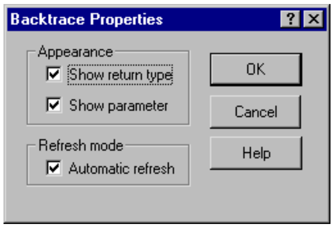
Figure 5-34 Backtrace processor view pop-up menu

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 To Memory** menu item functions as described in *Watch processor view pop-up menu* on page 5-25.

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 5-35, select **Properties...** from the pop-up menu.


Backtrace Properties

Appearance
☑ Show return type
☑ Show parameter

Refresh mode
☑ Automatic refresh

Figure 5-35 Backtrace Properties dialog

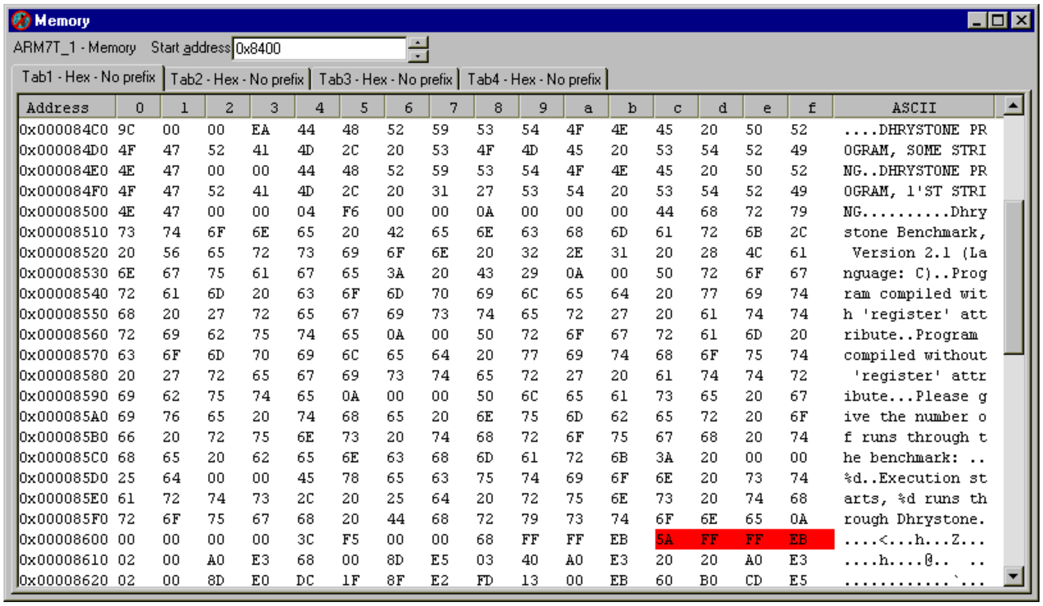
Refer to AXD online help for details of the other pop-up menu items.

5.4.5 Memory processor view

The Memory processor view allows 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 5-36.



```
Address     0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f   ASCII
0x000084C0 9C  00  00  EA  44  48  52  59  53  54  4F  4E  45  20  50  52   ....DHRYSTONE PR
0x000084D0 4F  47  52  41  4D  2C  20  53  4F  4D  45  20  53  54  52  49   OGRAM, SOME STRI
0x000084E0 4E  47  00  00  44  48  52  59  53  54  4F  4E  45  20  50  52   NG..DHRYSTONE PR
0x000084F0 4F  47  52  41  4D  2C  20  31  27  53  54  20  53  54  52  49   OGRAM, 1'ST STRI
0x00008500 4E  47  00  00  04  F6  00  00  0A  00  00  00  44  68  72  79   NG.........Dhry
0x00008510 73  74  6F  6E  65  20  42  65  6E  63  68  6D  61  72  6B  2C   stone Benchmark,
0x00008520 20  56  65  72  73  69  6F  6E  20  32  2E  31  20  28  4C  61    Version 2.1 (La
0x00008530 6E  67  75  61  67  65  3A  20  43  29  0A  00  50  72  6F  67   nguage: C)..Prog
0x00008540 72  61  6D  20  63  6F  6D  70  69  6C  65  64  20  77  69  74   ram compiled wit
0x00008550 68  20  27  72  65  67  69  73  74  65  72  27  20  61  74  74   h 'register' att
0x00008560 72  69  62  75  74  65  0A  00  50  72  6F  67  72  61  6D  20   ribute..Program
0x00008570 63  6F  6D  70  69  6C  65  64  20  77  69  74  68  6F  75  74   compiled without
0x00008580 20  27  72  65  67  69  73  74  65  72  27  20  61  74  74  72    'register' attr
0x00008590 69  62  75  74  65  0A  00  00  50  6C  65  61  73  65  20  67   ibute...Please g
0x000085A0 69  76  65  20  74  68  65  20  6E  75  6D  62  65  72  20  6F   ive the number o
0x000085B0 66  20  72  75  6E  73  20  74  68  72  6F  75  67  68  20  74   f runs through t
0x000085C0 68  65  20  62  65  6E  63  68  6D  61  72  6B  3A  20  00  00   he benchmark: ..
0x000085D0 25  64  00  00  45  78  65  63  75  74  69  6F  6E  20  73  74   %d..Execution st
0x000085E0 61  72  74  73  2C  20  25  64  20  72  75  6E  73  20  74  68   arts, %d runs th
0x000085F0 72  6F  75  67  68  20  44  68  72  79  73  74  6F  6E  65  0A   rough Dhrystone.
0x00008600 00  00  00  00  3C  F5  00  00  68  FF  FF  EB  5A  FF  FF  EB   ....<...h...Z...
0x00008610 02  00  A0  E3  68  00  8D  E5  03  40  A0  E3  20  20  A0  E3   ....h....@..  ..
0x00008620 02  00  8D  E0  DC  1F  8F  E2  FD  13  00  EB  60  B0  CD  E5   ............`...
```

Figure 5-36 Memory processor view

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 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 latter case, the ASCII characters corresponding to the 16 bytes are shown at the right of the line.

The four tabbed views 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 view is one page long, and starts at the address you specify in the Start Address box 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 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 grey-red if it is disabled. A watchpoint is highlighted in green, or grey-green if it is disabled.

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 4-15.

Memory processor view pop-up menu

To display the Memory pop-up menu, shown in Figure 5-37, right-click within the Memory processor view.

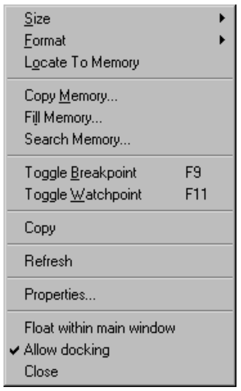


Figure 5-37 Memory processor view pop-up menu

The **Locate To Memory** menu item functions as described in *Watch processor view pop-up menu* on page 5-25.

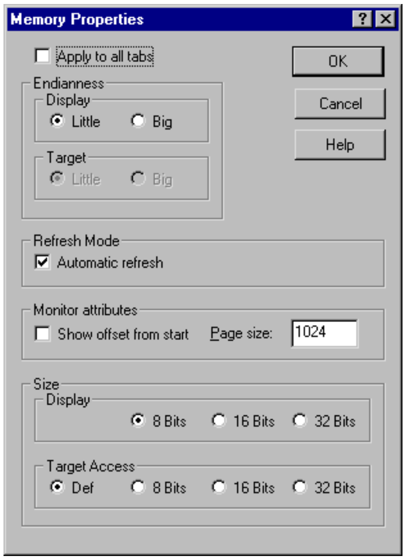
Select **Toggle Breakpoint** to toggle a breakpoint at the address defined by the current cursor position. If a breakpoint already exists at this address it is deleted. If no breakpoint exists at this address a default breakpoint is created here.

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.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

Refer to AXD online help for details of the other Memory pop-up menu items, including the Memory Properties dialog, shown in Figure 5-38.



[ ] Apply to all tabs

Endianness — Display: [✓ Little] Big; Target: [✓ Little] Big

Refresh Mode — [✓ Automatic refresh]

Monitor attributes — [ ] Show offset from start; Page size: 1024

Size — Display: [✓ 8 Bits] 16 Bits 32 Bits; Target Access: [✓ Def] 8 Bits 16 Bits 32 Bits

Figure 5-38 Memory Properties dialog

Data width for memory reads and writes

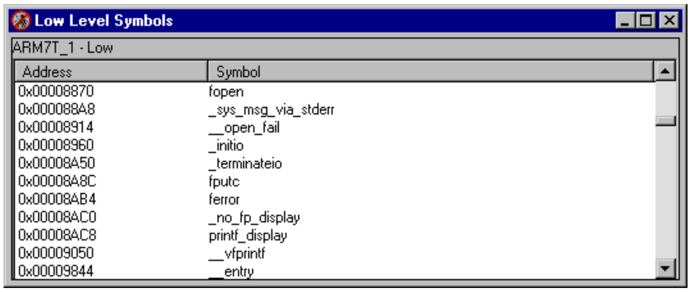
The Target Access group of radio buttons in the Memory Properties dialog allows you to specify the width of data read from or written to memory. Unless you have a particular requirement, use the **Def** setting to indicate that you want the debugger to decide.

5.4.6 Low Level Symbols processor view

The Low Level Symbols processor view allows 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 5-39.



Address	Symbol
0x00008870	fopen
0x000088A8	_sys_msg_via_stderr
0x00008914	__open_fail
0x00008960	_initio
0x00008A50	_terminateio
0x00008A8C	fputc
0x00008AB4	ferror
0x00008AC0	_no_fp_display
0x00008AC8	printf_display
0x00009050	__vfprintf
0x00009844	__entry

Figure 5-39 Low Level Symbols processor view

The left column shows addresses and the right column shows symbol strings. Use the pop-up menu to change the list between address order and 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 5-40, right-click within the Low Level Symbols processor view.

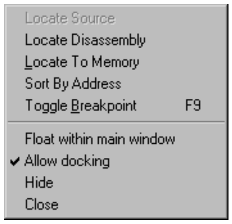


Figure 5-40 Low Level Symbols processor view pop-up menu

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.

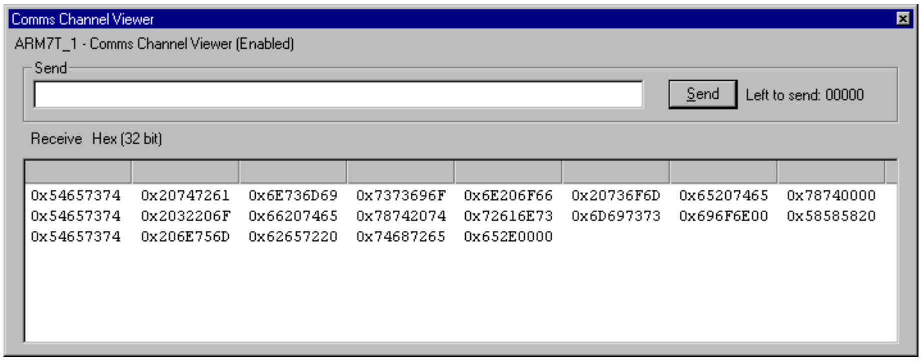
The **Locate To Memory** menu item functions as described in *Watch processor view pop-up menu* on page 5-25.

Refer to AXD online help for more details of these menu items.

5.4.7 Comms Channel processor view

The Comms Channel processor view allows 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 processor view is shown in Figure 5-41. You can enable or disable the communications channel viewer by checking or clearing a checkbox on the Processor Properties dialog (see *Configure Processor...* on page 5-91 or *Control system view pop-up menus* on page 5-46).



```
0x54657374  0x20747261  0x6E736D69  0x7373696F  0x6E206F66  0x20736F6D  0x65207465  0x78740000
0x54657374  0x2032206F  0x66207465  0x78742074  0x72616E73  0x6D697373  0x696F6E00  0x58585820
0x54657374  0x206E756D  0x62657220  0x74687265  0x652E0000
```

Figure 5-41 Comms Channel Viewer processor view

Use the Send area 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 channel viewer is displayed with the Auto-Toggle format. This converts the information into ASCII character codes and displays it in the Receive window, if the channel viewer is active. However, if `0xFFFFFFFF` is received, the Auto-Toggle format displays the following word as a number.

You can display received information in other formats, as described in *Comms Channel Viewer pop-up menu* on page 5-37.

Comms Channel Viewer pop-up menu

To display the Comms Channel pop-up menu (see Figure 5-42), right-click anywhere in the Comms Channel processor view except in the Send edit area or the Receive pane.

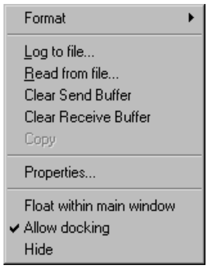


Figure 5-42 Comms Channel Viewer pop-up menu

Select **Format** to display the submenu shown in Figure 5-43.

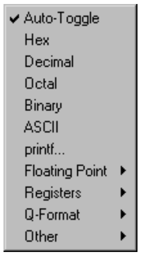
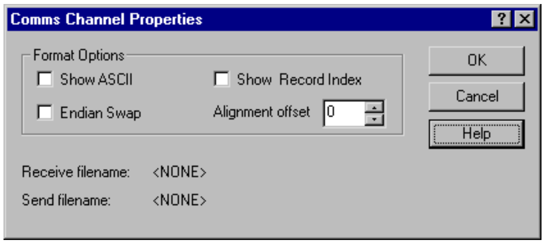


Figure 5-43 Comms channel viewer formats

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 alter the column widths by using the mouse to drag the column header dividers to the left or right. Select **Properties...** to display the dialog shown in Figure 5-44.


Comms Channel Properties

Format Options

Show ASCII [ ]
Show Record Index [ ]
Endian Swap [ ]
Alignment offset 0

Receive filename: <NONE>
Send filename: <NONE>

Figure 5-44 Comms channel viewer properties dialog

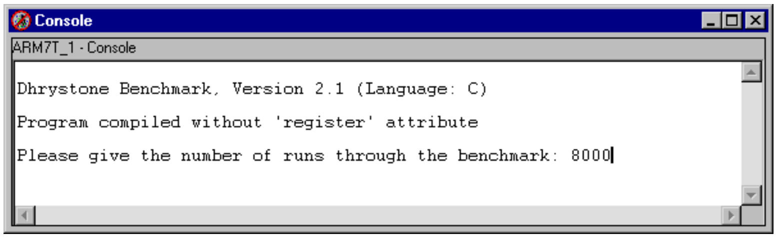
AXD online help describes this and all the Comms channel viewer pop-up menu items.

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 allows 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 5-45.



```
Dhrystone Benchmark, Version 2.1 (Language: C)
Program compiled without 'register' attribute
Please give the number of runs through the benchmark: 8000
```

Figure 5-45 Console processor view

Console processor view pop-up menu

To display the Console pop-up menu, shown in Figure 5-46, right-click within the Console processor view.

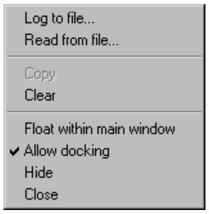


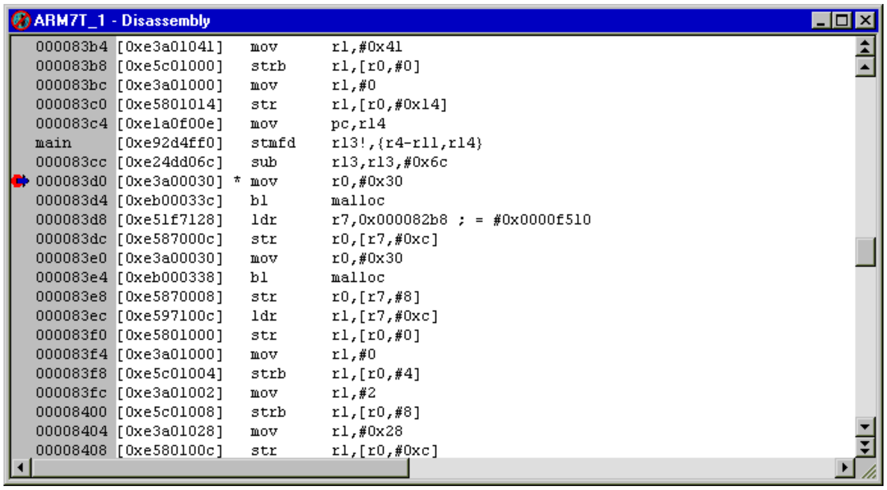
Figure 5-46 Console processor view pop-up menu

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 5-47. 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 5-49 on page 5-41).



```
000083b4 [0xe3a01041]   mov     r1,#0x41
000083b8 [0xe5c01000]   strb    r1,[r0,#0]
000083bc [0xe3a01000]   mov     r1,#0
000083c0 [0xe5801014]   str     r1,[r0,#0x14]
000083c4 [0xe1a0f00e]   mov     pc,r14
main     [0xe92d4ff0]   stmfd   r13!,{r4-r11,r14}
000083cc [0xe24dd06c]   sub     r13,r13,#0x6c
000083d0 [0xe3a00030] * mov     r0,#0x30
000083d4 [0xeb00033c]   bl      malloc
000083d8 [0xe51f7128]   ldr     r7,0x000082b8 ; = #0x0000f510
000083dc [0xe587000c]   str     r0,[r7,#0xc]
000083e0 [0xe3a00030]   mov     r0,#0x30
000083e4 [0xeb000338]   bl      malloc
000083e8 [0xe5870008]   str     r0,[r7,#8]
000083ec [0xe597100c]   ldr     r1,[r7,#0xc]
000083f0 [0xe5801000]   str     r1,[r0,#0]
000083f4 [0xe3a01000]   mov     r1,#0
000083f8 [0xe5c01004]   strb    r1,[r0,#4]
000083fc [0xe3a01002]   mov     r1,#2
00008400 [0xe5c01008]   strb    r1,[r0,#8]
00008404 [0xe3a01028]   mov     r1,#0x28
00008408 [0xe580100c]   str     r1,[r0,#0xc]
```

Figure 5-47 Disassembly processor view

Disassembly processor view pop-up menu

To display the Disassembly pop-up menu, shown in Figure 5-48, right-click within the Disassembly processor view.

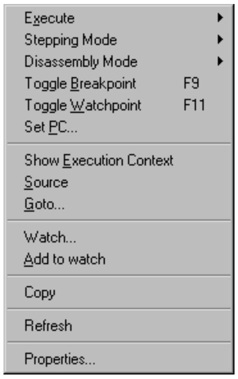


Figure 5-48 Disassembly processor view pop-up menu

To display a submenu duplicating the items you are most likely to want from the Execute main menu, point to **Execute** on the pop-up menu. See *Execute menu* on page 5-70 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, point to **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, point to **Stepping Mode** on the pop-up menu. The stepping modes available are:

- **Disassembly**, to step always in disassembly instructions
- **Strong source**, to step always in source code statements
- **Weak source**, to step in source code statements if available, otherwise in disassembly instructions (this is the default setting).

To display a submenu allowing you to change the setting of the code used for disassembly, point to **Disassembly Mode** on the pop-up menu.

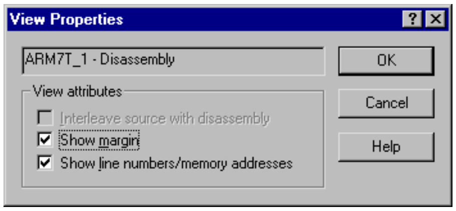
To set or delete a breakpoint at the current cursor position, select **Toggle Breakpoint** from the pop-up menu. To set or delete a watchpoint on a currently selected value, select **Toggle Watchpoint** from the pop-up menu.

To reset the program counter so that the instruction at the current cursor position is the next instruction to be executed, select **Set PC** from the pop-up menu.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

To display the dialog shown in Figure 5-49, select **Properties...** from the pop-up menu.

Refer to AXD online help for more details of all the items on the pop-up menu.



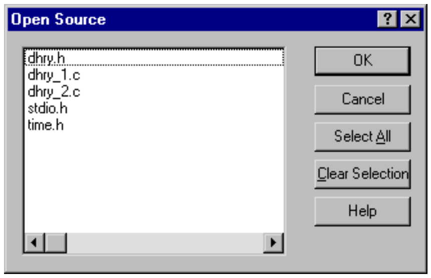
ARM7T_1 - Disassembly

View attributes
- [ ] Interleave source with disassembly
- [x] Show margin
- [x] Show line numbers/memory addresses

Figure 5-49 Disassembly Properties dialog

5.4.10 Source... processor view

The Source... processor view first displays a file selection dialog, similar to that shown in Figure 5-50.



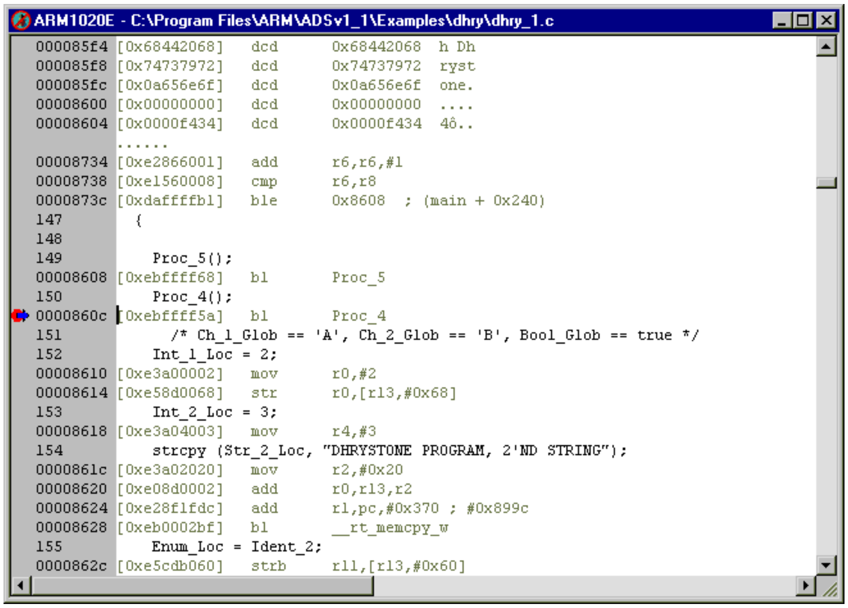
dhry.h
dhry_1.c
dhry_2.c
stdio.h
time.h

Figure 5-50 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, as shown in Figure 5-51 on page 5-42. If the file is not in the expected place, another dialog allows you to specify where it is or browse to find it.

Figure 5-51 on page 5-42 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.



```
ARM1020E - C:\Program Files\ARM\ADSv1_1\Examples\dhry\dhry_1.c
000085f4 [0x68442068]   dcd       0x68442068  h Dh
000085f8 [0x74737972]   dcd       0x74737972  ryst
000085fc [0x0a656e6f]   dcd       0x0a656e6f  one.
00008600 [0x00000000]   dcd       0x00000000  ....
00008604 [0x0000f434]   dcd       0x0000f434  4ô..
         ......
00008734 [0xe2866001]   add       r6,r6,#1
00008738 [0xe1560008]   cmp       r6,r8
0000873c [0xdaffffb1]   ble       0x8608  ; (main + 0x240)
147        {
148
149          Proc_5();
00008608 [0xebffff68]   bl        Proc_5
150          Proc_4();
0000860c [0xebffff5a]   bl        Proc_4
151            /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
152          Int_1_Loc = 2;
00008610 [0xe3a00002]   mov       r0,#2
00008614 [0xe58d0068]   str       r0,[r13,#0x68]
153          Int_2_Loc = 3;
00008618 [0xe3a04003]   mov       r4,#3
154          strcpy (Str_2_Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
0000861c [0xe3a02020]   mov       r2,#0x20
00008620 [0xe08d0002]   add       r0,r13,r2
00008624 [0xe28f1fdc]   add       r1,pc,#0x370 ; #0x899c
00008628 [0xeb0002bf]   bl        __rt_memcpy_w
155          Enum_Loc = Ident_2;
0000862c [0xe5cdb060]   strb      r11,[r13,#0x60]
```

Figure 5-51 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 Source pop-up menu, shown in Figure 5-52, right-click within the Source... processor view.

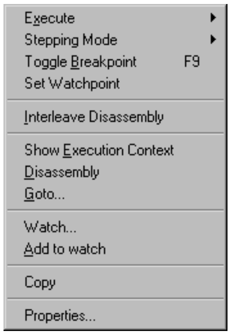


Figure 5-52 Source... processor view pop-up menu

To display a submenu duplicating the items you are most likely to need from the Execute main menu, point to **Execute** on the pop-up menu. See *Execute menu* on page 5-70 for details of all items except **Set Next Statement**.

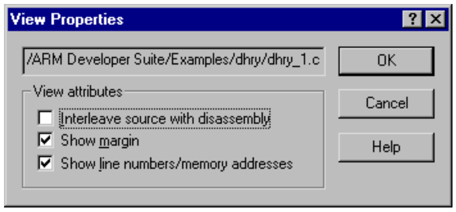
Set Next Statement is the item that appears on the Execute submenu and not in the Execute main menu. To resume execution at a specific statement, without executing any intervening statements, right-click on the required statement in the Source... processor view, point to **Execute** in the pop-up menu, and select **Set Next Statement**.

To display a submenu allowing you to change the setting of the stepping mode, point to **Stepping Mode** on the pop-up menu. The stepping modes available are:

- **Disassembly**, to step always in disassembly instructions
- **Strong source**, to step always in source code statements
- **Weak source**, to step in source code statements if available, otherwise in disassembly instructions (this is the default setting).

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 5-53, select **Properties...** from the pop-up menu.


View Properties

/ARM Developer Suite/Examples/dhry/dhry_1.c

View attributes
- [ ] Interleave source with disassembly
- [x] Show margin
- [x] Show line numbers/memory addresses

Figure 5-53 Source View Properties dialog

Refer to AXD online help for more details of all the items on this pop-up menu.

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 5-54 shows an example of a System Views menu.

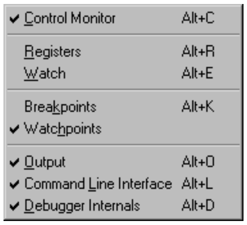


Figure 5-54 System Views menu

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:

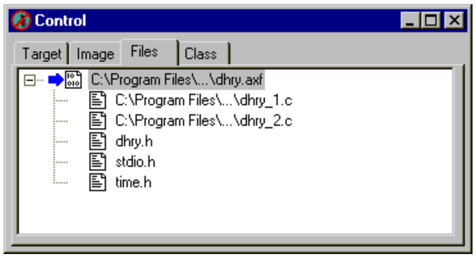
- *Control system view* on page 5-45
- *Registers system view* on page 5-51
- *Watch system view* on page 5-53
- *Breakpoints system view* on page 5-55
- *Watchpoints system view* on page 5-58
- *Output system view* on page 5-60
- *Command Line Interface system view* on page 5-61
- *Debugger Internals system view* on page 5-65.

5.5.1 Control system view

The Control system view shows details of all current processors, and allows you to examine this information in several ways. Tabbed pages available are:

- Target
- Image
- Files
- Class.

Figure 5-55 shows a Control system view with its Files tab selected.



```
C:\Program Files\...\dhry.axf
    C:\Program Files\...\dhry_1.c
    C:\Program Files\...\dhry_2.c
    dhry.h
    stdio.h
    time.h
```

Figure 5-55 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. Any processor with a coprocessor has its coprocessor shown as a child.

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.

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 tabbed page, you can select a processor
- on the Image tabbed page, you can select an image or a processor
- on the Files tabbed page, you can select an image or a file
- on the Class tabbed page, 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 5-56.

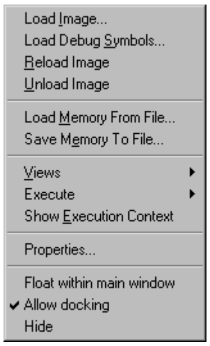
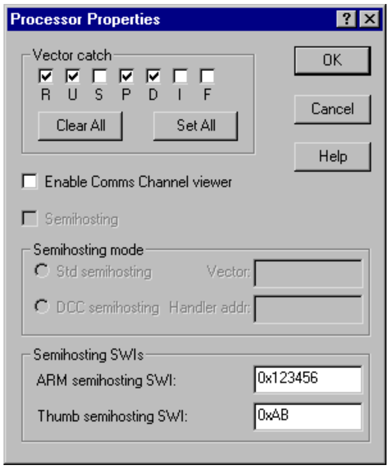


Figure 5-56 Pop-up menu when a processor is selected

Select **Properties...** from this pop-up menu to display the dialog shown in Figure 5-57.



Vector catch: R ✓, U ✓, S ☐, P ✓, D ✓, I ☐, F ☐

☐ Enable Comms Channel viewer

☐ Semihosting

Semihosting mode
○ Std semihosting Vector:
○ DCC semihosting Handler addr:

Semihosting SWIs
ARM semihosting SWI: 0x123456
Thumb semihosting SWI: 0xAB

Figure 5-57 Processor Properties dialog

For a description of this dialog, see *Configure Processor...* on page 5-91.

Image pop-up menu

With an image selected, the pop-up menu is as shown in Figure 5-58.

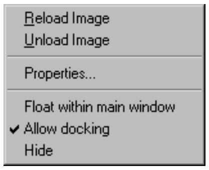


Figure 5-58 Pop-up menu when an image is selected

If you select **Properties...** from this pop-up menu, the dialog shown in Figure 5-59 is displayed.

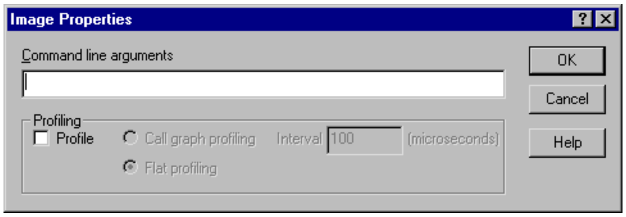


Figure 5-59 Image Properties dialog

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 5-60.

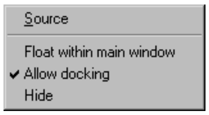


Figure 5-60 Pop-up menu when a file is selected

Select **Source** from this pop-up menu to display a Source processor view, showing the source code associated with the selected file.

Function pop-up menu

With a function selected, the pop-up menu is as shown in Figure 5-61.

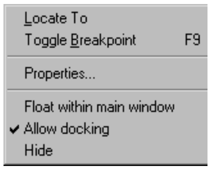
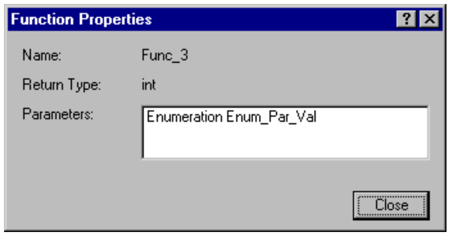


Figure 5-61 Pop-up menu when a function is selected

Select **Properties...** from this pop-up menu to display the dialog shown in Figure 5-62.



Name: Func_3
Return Type: int
Parameters: Enumeration Enum_Par_Val

Figure 5-62 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 5-63.

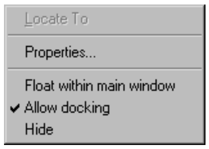
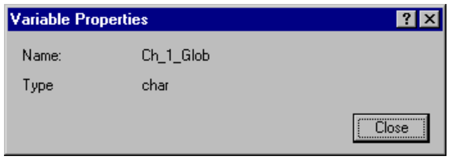


Figure 5-63 Pop-up menu when a variable is selected

If you select **Properties...** from this pop-up menu, the dialog shown in Figure 5-64 is displayed.


Variable Properties

Name: Ch_1_Glob

Type char

Figure 5-64 Variable Properties dialog

The Variable Properties dialog shows the name and type of the variable.

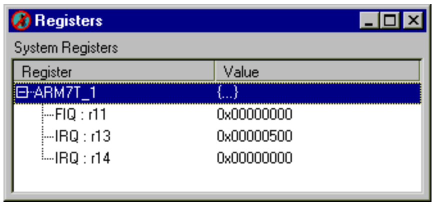
5.5.2 Registers system view

The Registers system view can display registers from more than one processor. It also allows 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 5-65 shows a typical Registers system view.



Register	Value
⊟ ARM7T_1	{...}
FIQ : r11	0x00000000
IRQ : r13	0x00000500
IRQ : r14	0x00000000

Figure 5-65 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.

Registers system view pop-up menu

To display the Registers pop-up menu, shown in Figure 5-66, right-click within the Registers system view.

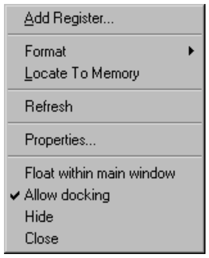


Figure 5-66 Registers system view pop-up menu

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 4-15 for details of the formats available, and to AXD online help for other details of the Registers pop-up menu items.

To add a register from any processor to those displayed in a Registers system view, select **Add Register** from the pop-up menu.

The **Locate To Memory** menu item functions as described in *Watch processor view pop-up menu* on page 5-25.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

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.

5.5.3 Watch system view

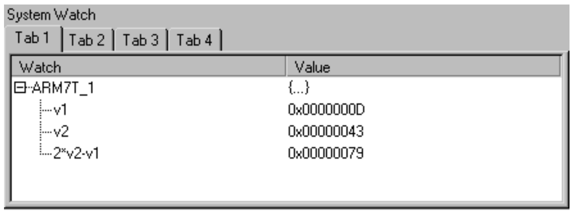
The Watch system view allows you to examine the value of variables, or of expressions depending on variables, in the images associated with various processors. You might require several processor views to see what you can display in a single system view.

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 a line to this view is to right-click on an item in a Variables processor view 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 5-69 on page 5-54).

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 5-67.



Watch	Value
⊟ ARM7T_1	{...}
v1	0x0000000D
v2	0x00000043
2*v2-v1	0x00000079

Figure 5-67 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

To display the Watch pop-up menu, shown in Figure 5-68, right-click within the Watch system view.

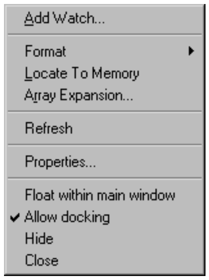
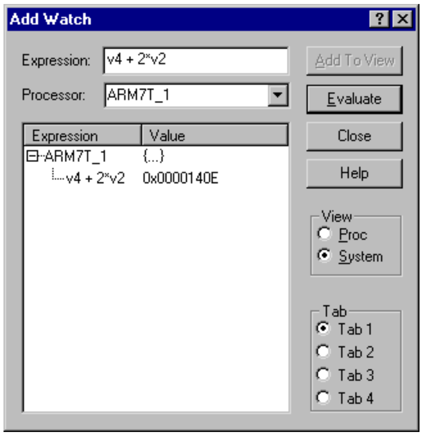


Figure 5-68 Watch system view pop-up menu

To display the dialog shown in Figure 5-69, select **Add Watch** from the pop-up menu.



Expression	Value
⊟ ARM7T_1	{...}
v4 + 2*v2	0x0000140E

Figure 5-69 Add Watch dialog

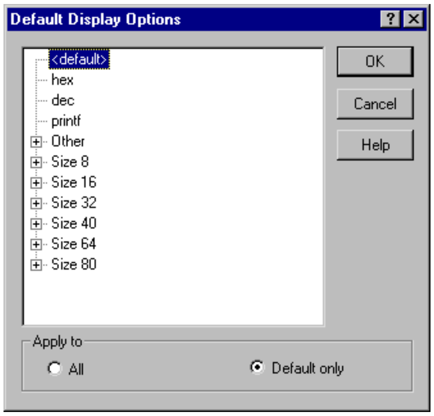
Enter a new expression to watch. Specify the processor, whether the new watch must be added to the Watch processor view or system view, and on which tabbed page it must appear. Figure 5-69 shows Tab 1 of the System view as the chosen destination. Select an expression and click the **Evaluate** button to see the result of its evaluation.

To add the selected expression to the chosen view, click the **Add To View** button.

The **Locate To Memory** menu item functions as described in *Watch processor view pop-up menu* on page 5-25.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

To display the dialog shown in Figure 5-70, select **Properties...** from the pop-up menu.



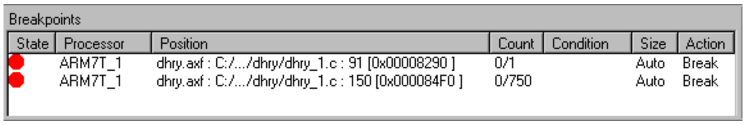
Apply to: All [✓ Default only]

Figure 5-70 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 5-71, allows you to set, modify, or remove breakpoints. You can alter the column widths by dragging the dividing lines between the column headings to the left or right.



Breakpoints

State	Processor	Position	Count	Condition	Size	Action
●	ARM7T_1	dhry.axf : C:/.../dhry/dhry_1.c : 91 [0x00008290]	0/1		Auto	Break
●	ARM7T_1	dhry.axf : C:/.../dhry/dhry_1.c : 150 [0x000084F0]	0/750		Auto	Break

Figure 5-71 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 centre of the disc becomes grey. 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 5-72 on page 5-56 and select **Add**.

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**.

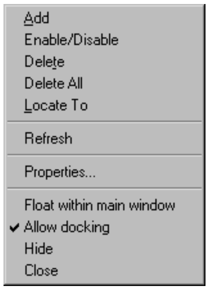
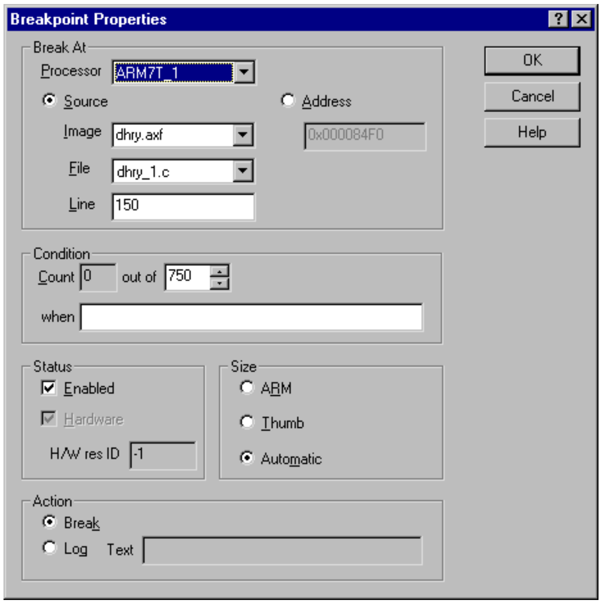


Figure 5-72 Breakpoints system view pop-up menu

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

Whether you are adding a new breakpoint or modifying an existing breakpoint, you use the Breakpoint Properties dialog shown in Figure 5-73.



Breakpoint Properties

Break At
Processor: ARM7T_1
[✓ Source] Address
Image: dhry.axf
File: dhry_1.c
Line: 150
Address: 0x000084F0

Condition
Count: 0 out of 750
when:

Status
[✓] Enabled
[✓] Hardware
H/W res ID: -1

Size
ARM
Thumb
[✓ Automatic]

Action
[✓ Break]
Log Text:

OK | Cancel | Help

Figure 5-73 Breakpoint Properties dialog

The Break At fields 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 Condition fields allow 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.

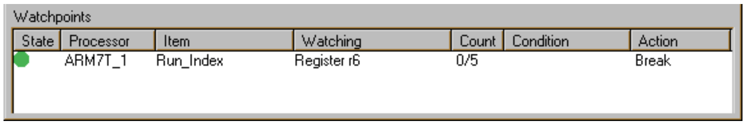
You are recommended to leave the Size set to Automatic, but you can change this to ARM (32-bit) or Thumb (16-bit) if necessary. For example, the debugger might not be able to determine whether it is debugging ARM code or Thumb code if:

- the project was built without debugging information (`-g-` switches off debugging)
- you are debugging a ROM image.

The setting in the Action box 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 5-60). The pop-up menu of the Output system view allows 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 allows you to set, modify, or remove watchpoints. The resulting dialog is shown in Figure 5-74. You can alter the column widths by dragging the dividing lines between the column headings to the left or right.



Watchpoints

State	Processor	Item	Watching	Count	Condition	Action
●	ARM7T_1	Run_Index	Register r6	0/5		Break

Figure 5-74 Watchpoints dialog

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 centre of the disc becomes grey. 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 5-75 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**.

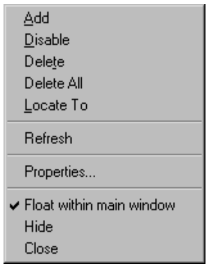
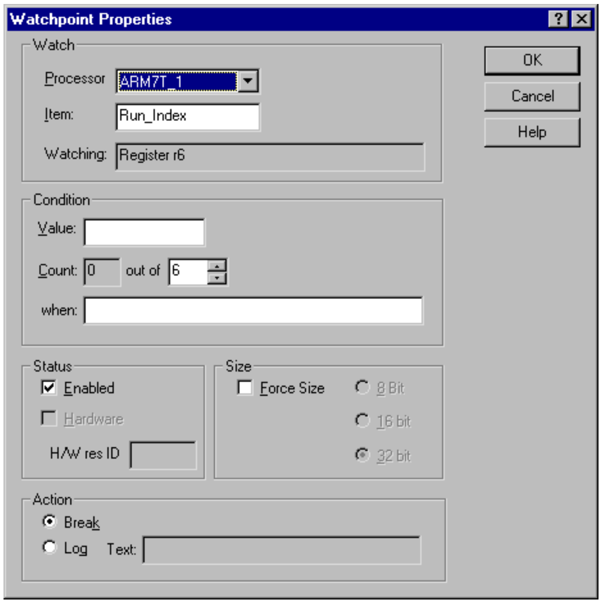


Figure 5-75 Watchpoints system view pop-up menu

Select **Locate To** to display a Disassembly processor view containing the selected watchpoint.

Select **Refresh** to update and recalculate the displayed data values. This item is useful only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

Whether you are adding a new watchpoint or modifying an existing watchpoint, you use the Watchpoint Properties dialog shown in Figure 5-76 on page 5-59.



Watchpoint Properties

Watch
Processor: ARM7T_1
Item: Run_Index
Watching: Register r6

Condition
Value:
Count: 0 out of 6
when:

Status
[✓ Enabled]
Hardware
H/W res ID

Size
Force Size
8 Bit
16 bit
[✓ 32 bit]

Action
[✓ Break]
Log Text:

Figure 5-76 Watchpoint Properties dialog

The Watch fields specify the location of the watched value. The Processor field allows 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.

The Condition fields allow 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 a 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.

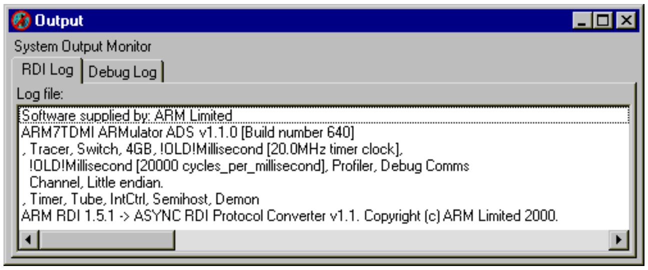
Under Size, you are recommended to leave **Force Size** unchecked. The area of memory watched is then the size of the variable if you are watching a variable, or a 4-byte word if you are watching a memory location. If you force the size of the watched area of memory you can select 8, 16, or 32 bits.

The setting in the Action box 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 5-60). The pop-up menu of the Output system view allows 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.

Select **Output** from the System Views menu to display a window, shown in Figure 5-77, containing two tabbed pages, labeled RDI Log and Debug Log.



```
Software supplied by: ARM Limited
ARM7TDMI ARMulator ADS v1.1.0 [Build number 640]
, Tracer, Switch, 4GB, !OLD!Millisecond [20.0MHz timer clock],
  !OLD!Millisecond [20000 cycles_per_millisecond], Profiler, Debug Comms
  Channel, Little endian.
, Timer, Tube, IntCtrl, Semihost, Demon
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter v1.1. Copyright (c) ARM Limited 2000.
```

Figure 5-77 Output system view

Click **RDI Log** 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 **Debug Log** 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 5-55). 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 5-78, right-click on either the RDI Log page or the Trace page.

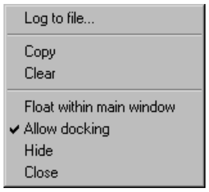


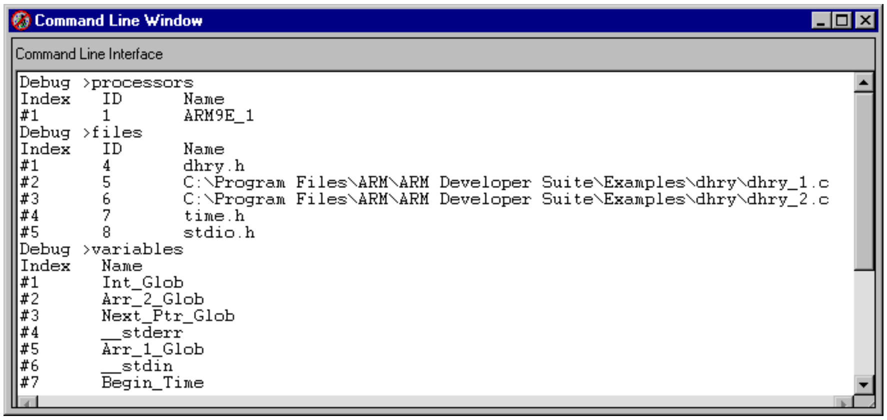
Figure 5-78 Output system view pop-up menu

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.

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 5-79. Any data that you request is displayed in the CLI system view.



```
Debug >processors
Index   ID      Name
#1      1       ARM9E_1
Debug >files
Index   ID      Name
#1      4       dhry.h
#2      5       C:\Program Files\ARM\ARM Developer Suite\Examples\dhry\dhry_1.c
#3      6       C:\Program Files\ARM\ARM Developer Suite\Examples\dhry\dhry_2.c
#4      7       time.h
#5      8       stdio.h
Debug >variables
Index   Name
#1      Int_Glob
#2      Arr_2_Glob
#3      Next_Ptr_Glob
#4      __stderr
#5      Arr_1_Glob
#6      __stdin
#7      Begin_Time
```

Figure 5-79 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 5-80, right-click within the Command Line Interface system view.

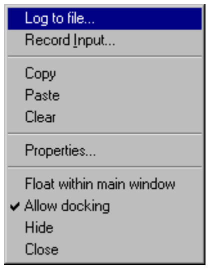


Figure 5-80 CLI system view pop-up menu

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.

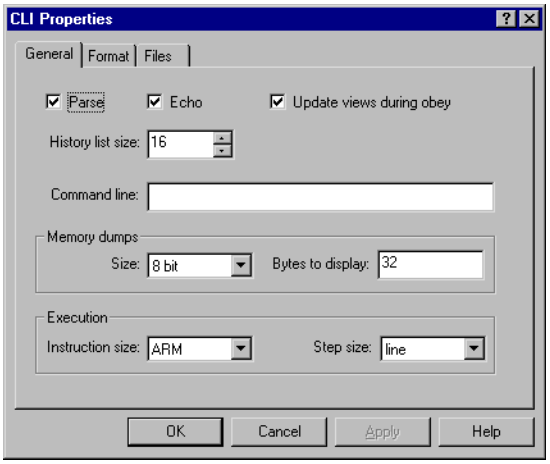
Clear enables you to clear the current contents of the CLI system view.

To display the dialogs shown in Figure 5-81 on page 5-63, Figure 5-82 on page 5-64, and Figure 5-83 on page 5-64, select **Properties...** from the pop-up menu. AXD online help gives details of all the fields in these dialogs, in addition to details of all the pop-up menu items.

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 5-81 on page 5-63 shows the CLI Properties dialog with the default General tab selected.



☑ Parse ☑ Echo ☑ Update views during obey

History list size: 16

Command line:

Memory dumps

Size: 8 bit

Bytes to display: 32

Execution

Instruction size: ARM

Step size: line

Figure 5-81 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 allows 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 checkbox 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 5-82 on page 5-64.

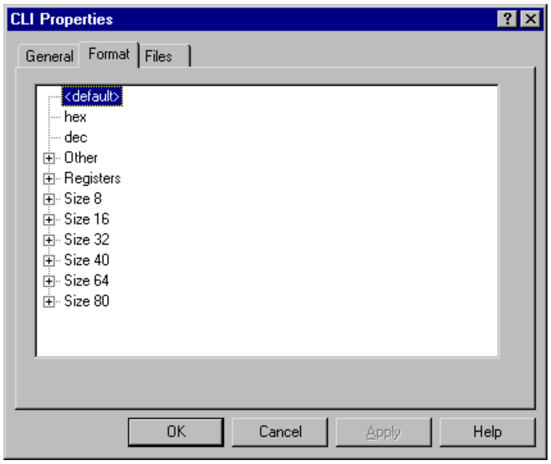
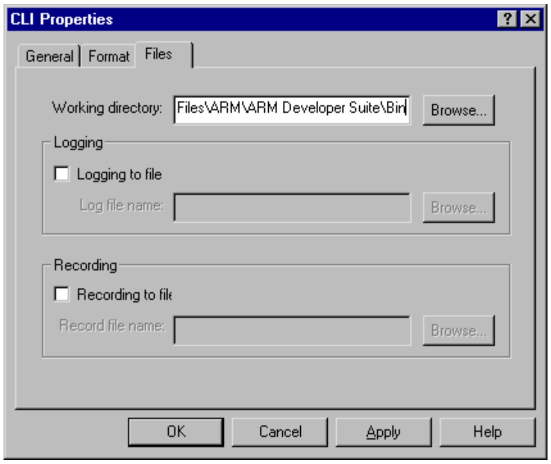


Figure 5-82 Command Line Interface Properties dialog, Format tab

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 allows you to choose one.

Click on the Files tab to display the dialog shown in Figure 5-83.



Working directory: Files\ARM\ARM Developer Suite\Bin

Figure 5-83 Command Line Interface Properties dialog, Files tab

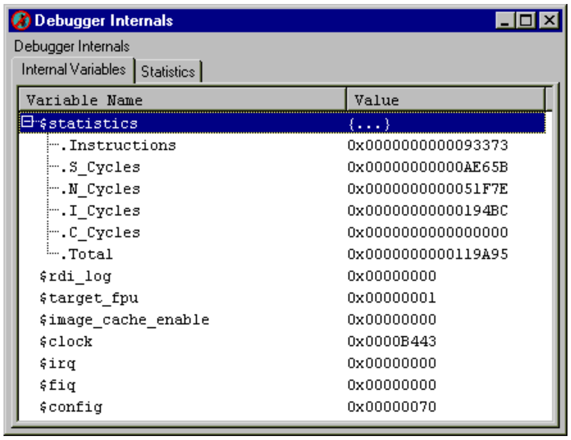
5.5.8 Debugger Internals system view

The Debugger Internals system view has two tabbed pages:

- *Internal Variables* on page 5-65
- *Statistics* on page 5-68.

Internal Variables

The first tabbed page of the Debugger Internals system view shows Internal Variables, as shown in Figure 5-84.



```
Variable Name            Value
⊟$statistics             {...}
  .Instructions          0x0000000000093373
  .S_Cycles              0x00000000000AE65B
  .N_Cycles              0x0000000000051F7E
  .I_Cycles              0x00000000000194BC
  .C_Cycles              0x0000000000000000
  .Total                 0x0000000000119A95
$rdi_log                 0x00000000
$target_fpu              0x00000001
$image_cache_enable      0x00000000
$clock                   0x0000B443
$irq                     0x00000000
$fiq                     0x00000000
$config                  0x00000070
```

Figure 5-84 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 following are displayed on the Internal Variables tabbed page of the Debugger Internals system view:

$statistics This is a group of internal variables that you can examine more clearly on the Statistics tabbed page (see *Statistics* on page 5-68) or by using a CLI command (see *statistics* on page 6-49).

$rdi_log The two least significant bits have the following meanings:

Bit 0	RDI (0 = off, 1 = on).
Bit 1	Device Driver Logging (0 = off, 1 = on).

$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:

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.

All other values are invalid and result in `$target_fpu` being set to 1.

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 Compiler, Linker, and Utilities Guide*.

$image_cache_enable

This flag is useful only when you are debugging a target that must not stop execution. It indicates that some debug information that might otherwise be unavailable is held in the host computer memory.

$clock This variable applies to ARMulator only and contains the number of microseconds that have elapsed since the application program began execution. The value is based on the ARMulator clock speed setting, and is unavailable if that speed is set to 0.00 (see also *Configure Target...* on page 5-81). 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 pane of the Control system view (for example, `ARM1020E`).

<var_name>	is the name of the variable, and can include:	
	`irq`	(For example, `$ARM1020E$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, `$ARM1020E$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.

Your debug target might create other variables (for example, `$ARM1020E$other`). 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 4-15.

Debugger Internal Variables pop-up menu

Right-click inside the Debugger Internals system view with the Internal Variables page selected to display the pop-up menu shown in Figure 5-85.

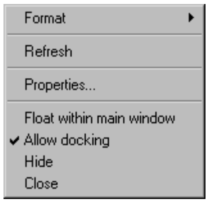


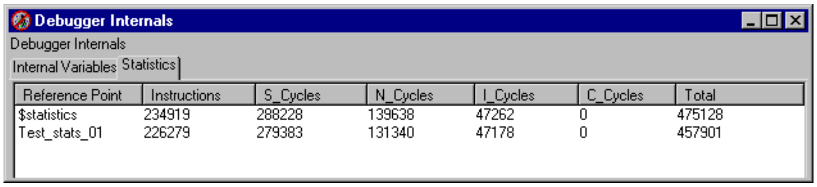
Figure 5-85 Internal Variables pop-up menu

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 only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

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 5-86.



Reference Point	Instructions	S_Cycles	N_Cycles	I_Cycles	C_Cycles	Total
$statistics	234919	288228	139638	47262	0	475128
Test_stats_01	226279	279383	131340	47178	0	457901

Figure 5-86 Debugger Internals, Statistics

A group of debugger internal variables contains statistics relating to your current debugging session. These variables are displayed more clearly on the Statistics tabbed page than on the Internal Variables tabbed page. Drag the column divider lines to the left or right to alter 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 6-49).

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 tabbed page of the Debugger Internals system view, and select **Add New Reference Point**.

The following information is displayed:

Reference Point

The name you specify to identify each line of statistics that you add.

Instructions The number of program instructions executed.

S-Cycles Sequential cycles. The CPU requests transfer to or from the same address, or an address a word or halfword after the preceding address.

N-Cycles Nonsequential cycles. The CPU requests transfer to or from an address that is unrelated to the address used in the preceding cycle.

I-Cycles Internal cycles. The CPU does not require a transfer because it is performing an internal function (or running from cache).

C-Cycles Coprocessor cycles.

Total The sum of the counts of S-cycles, N-cycles, I-cycles, and C-cycles.

If you use a map file (see *ARMulator configuration* on page 5-82) the display shows additional information, including:

Wait_States	The number of wait-states added by the Mapfile component. It is only displayed if you set the CountWaitStates configuration flag to True.
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

——— **Note** ———

When simulating Harvard architecture cores such as ARM9TDMI™ and StrongARM®, different statistics are accumulated. In these cases, the meanings are:

N-Cycles	Cycles in which an instruction was fetched and no data was fetched.
S-Cycles	Cycles in which an instruction was fetched and data was fetched.
I-Cycles	Cycles in which no instruction was fetched and no data was fetched.
C-Cycles	Cycles in which no instruction was fetched and data was fetched.

Statistics pop-up menu

Right-click inside the Debugger Internals system view with the Statistics page selected to display the pop-up menu shown in Figure 5-87.

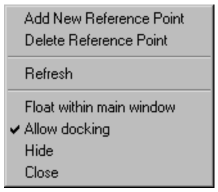


Figure 5-87 Statistics pop-up menu

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 only if the target supports RealMonitor. See also *Refresh All* on page 5-95.

5.6 Execute menu

The Execute menu (see Figure 5-88), lets you control how execution continues from the current point.

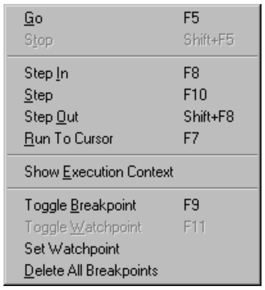


Figure 5-88 Execute menu

The Execute menu items are described under the following headings:

- *Go* on page 5-70
- *Stop* on page 5-70
- *Step In* on page 5-71
- *Step* on page 5-71
- *Step Out* on page 5-71
- *Run To Cursor* on page 5-71
- *Show Execution Context* on page 5-71
- *Toggle Breakpoint* on page 5-71
- *Toggle Watchpoint* on page 5-72
- *Set Watchpoint* on page 5-72
- *Delete All Breakpoints* on page 5-72.

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.

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.

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.

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 Compiler, Linker, and Utilities Guide*.

5.6.5 Step Out

This completes execution of the current function and stops when control returns to the caller.

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.

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.

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.

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.

5.6.11 Delete All Breakpoints

To delete all currently set breakpoints, select **Delete All Breakpoints** from the Execute menu.

5.7 Options menu

The Options menu, shown in Figure 5-89, allows you to examine and change a variety of settings, including some that affect the appearance of the debugger screen. This menu also allows you to start and stop profiling.

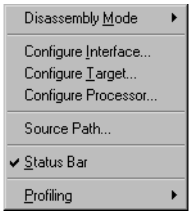


Figure 5-89 Options menu

The Options menu items are described under the following headings:

- *Disassembly Mode* on page 5-73
- *Configure Interface...* on page 5-74
- *Configure Target...* on page 5-81
- *Configure Processor...* on page 5-91
- *Source Path...* on page 5-93
- *Status Bar* on page 5-93
- *Profiling* on page 5-93.

5.7.1 Disassembly Mode

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, or Thumb. One of the 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
- Formatting
- Session File
- Toolbars
- Timed Refresh.

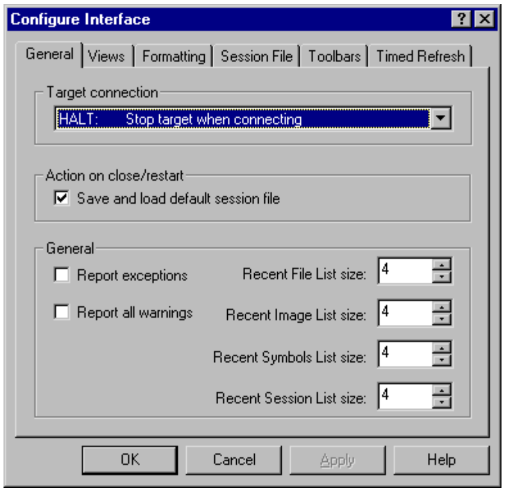
To display detailed information about the features of the currently displayed tabbed page, click **Help**.

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 5-90 shows the General tabbed page of the Configure Interface dialog.



Target connection
HALT: Stop target when connecting

Action on close/restart
☑ Save and load default session file

General
☐ Report exceptions
☐ Report all warnings
Recent File List size: 4
Recent Image List size: 4
Recent Symbols List size: 4
Recent Session List size: 4

Figure 5-90 Configure Interface, General tab

The General tabbed page of the Configure Interface dialog allows 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 allows 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.

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 4-13 for more information.

Halt The debugger is allowed to, and does, stop execution of the target when the connection is made. 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

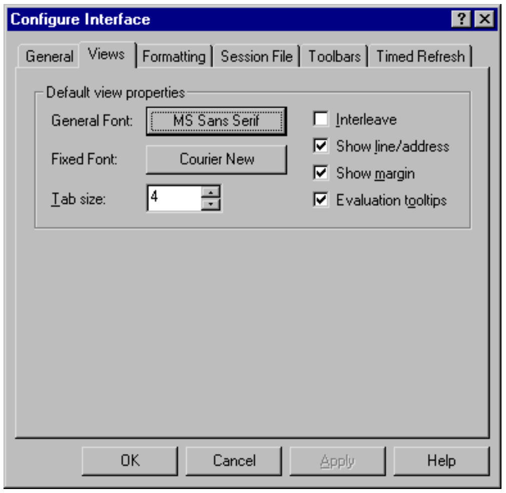
Attach If the target supports *RealMonitor* and a non-intrusive connection can be made, then the connection is made without stopping target execution. If the target does not support *RealMonitor* then the connection is still made even though doing so stops the target execution.

Under Action on close/restart you can select a **Save and load session file** checkbox. 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** checkbox is cleared, details are not saved at the end of the current session, and the next session begins in the usual default state.

The General check boxes control the types of messages recorded in the Debug Log page of the Output system view (see *Output system view* on page 5-60), and the List size fields allow you to control the amount of recent history that is maintained.

Views

Figure 5-91 shows the Views tabbed page of the Configure Interface dialog.


Configure Interface

General | Views | Formatting | Session File | Toolbars | Timed Refresh

Default view properties

General Font: MS Sans Serif

Fixed Font: Courier New

Tab size: 4

☐ Interleave
☑ Show line/address
☑ Show margin
☑ Evaluation tooltips

Figure 5-91 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.

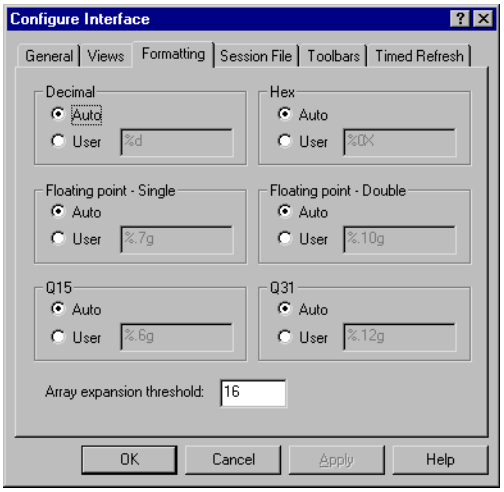
The Fixed Font you select applies to the following views:

- Command Line Interface
- Comms Channel
- Console
- Debugger Internals
- Disassembly
- Memory
- Registers
- Source
- Variables.

To display detailed information on all the fields and checkboxes on this tabbed page, click **Help**.

Formatting

Figure 5-92 shows the Formatting tabbed page of the Configure Interface dialog.



Configure Interface

General | Views | Formatting | Session File | Toolbars | Timed Refresh

Decimal: (●) Auto ( ) User %d

Hex: (●) Auto ( ) User %0X

Floating point - Single: (●) Auto ( ) User %.7g

Floating point - Double: (●) Auto ( ) User %.10g

Q15: (●) Auto ( ) User %.6g

Q31: (●) Auto ( ) User %.12g

Array expansion threshold: 16

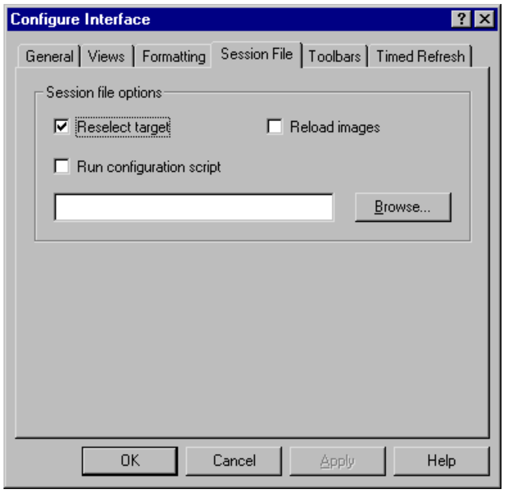
Figure 5-92 Configure Interface, Formatting tab

The Formatting tabbed page of the Configure Interface dialog allows 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.

Session File

Figure 5-93 shows the Session File tabbed page of the Configure Interface dialog. This tabbed page allows 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.



☑ Reselect target ☐ Reload images

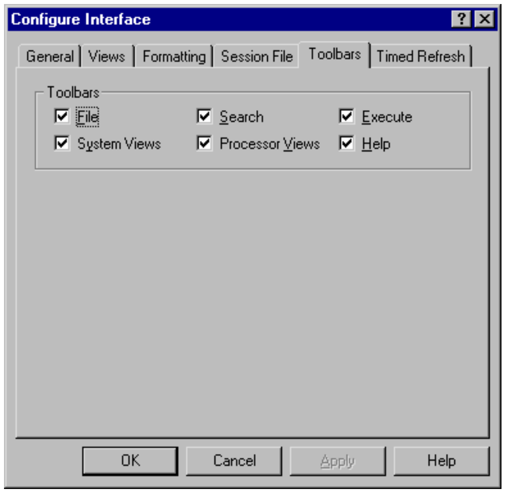
☐ Run configuration script

Figure 5-93 Configure Interface, Session File tab

Under Session file options, you can choose whether or not to **Reselect target**. If this is checked, 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 images into the target. If you use the **Browse** button to locate and select a script file, and check the **Run configuration script** checkbox, 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 5-94 shows the Toolbars tabbed page of the Configure Interface dialog.



Toolbars

- [✓] File
- [✓] Search
- [✓] Execute
- [✓] System Views
- [✓] Processor Views
- [✓] Help

Figure 5-94 Configure Interface, Toolbars tab

The Toolbars check boxes 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 5-95 shows the Timed Refresh tabbed page of the Configure Interface dialog.

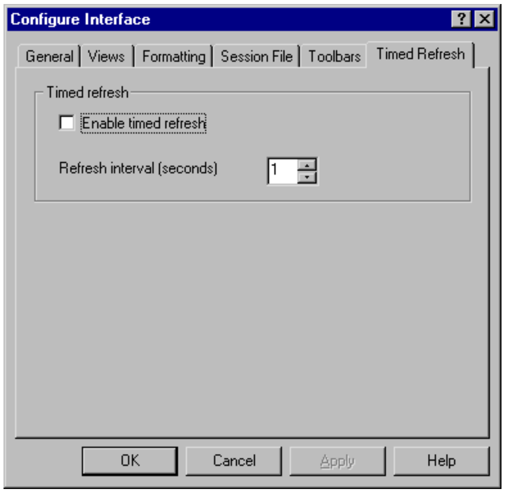


Figure 5-95 Configure Interface, Timed Refresh tab

The Timed Refresh tabbed page of the Configure Interface dialog is particularly useful when you are debugging a target that supports *RealMonitor* (see *RealMonitor support* on page 4-13).

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.

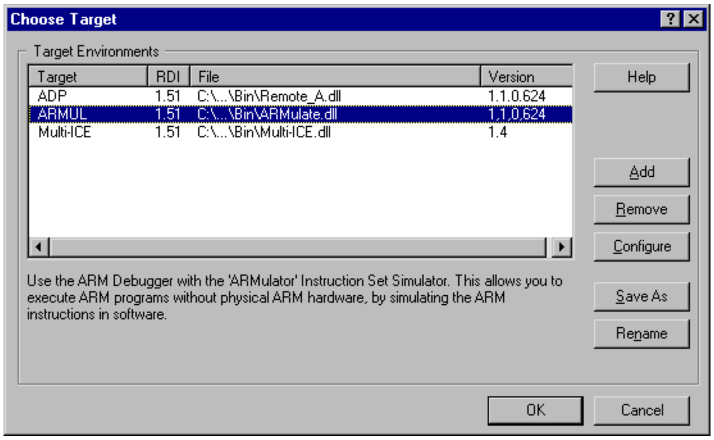
The pop-up menu available in most views includes a **Refresh** item, but that refreshes the information in that view only. The Windows menu includes a **Refresh All** item, to refresh all the displayed views at the same time.

If you select the **Enable Timed Refresh** checkbox on the Timed Refresh tabbed page of the Configure Interface dialog so that it is checked, then all displayed views are refreshed automatically and regularly. The same tabbed page allows you to set the Refresh Interval in seconds. An alternative way of checking or clearing the **Enable Timed Refresh** checkbox is by selecting the **Timed Refresh** item on the Windows menu.

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 allows 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 5-96.



Target	RDI	File	Version
ADP	1.51	C:\...\Bin\Remote_A.dll	1.1.0.624
ARMUL	1.51	C:\...\Bin\ARMulate.dll	1.1.0.624
Multi-ICE	1.51	C:\...\Bin\Multi-ICE.dll	1.4

Use the ARM Debugger with the 'ARMulator' Instruction Set Simulator. This allows you to execute ARM programs without physical ARM hardware, by simulating the ARM instructions in software.

Figure 5-96 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. When you can see the target you want in the list, select that line, and click the **Configure** button.

The appearance of the configuration dialog depends on the target you selected. Examples follow showing:

- *ARMulator configuration* on page 5-82
- *Multi-ICE configuration* on page 5-84
- *Remote_A configuration* on page 5-85
- *Gateway configuration* on page 5-88.

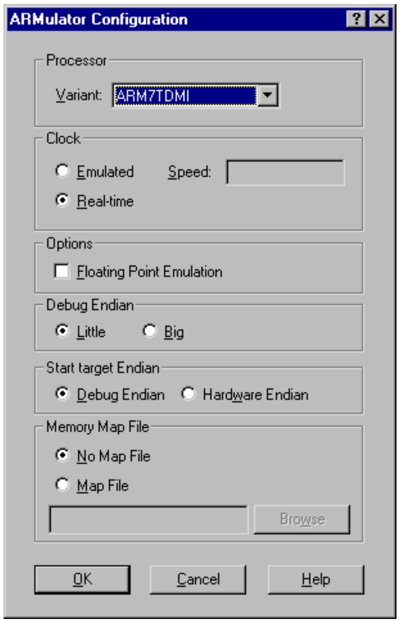
——— **Note** ———

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 are 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 5-97.


ARMulator Configuration

Processor
Variant: ARM7TDMI

Clock
( ) Emulated Speed:
(•) Real-time

Options
[ ] Floating Point Emulation

Debug Endian
(•) Little ( ) Big

Start target Endian
(•) Debug Endian ( ) Hardware Endian

Memory Map File
(•) No Map File
( ) Map File

Figure 5-97 ARMulator configuration dialog

The ARMulator Configuration dialog allows you to examine and change the following settings:

Processor	Specify which ARM processor you want ARMulator to simulate.
Clock	Choose between simulating a processor clock running at a speed that you can specify, or executing instructions in real time.
Options	Specify whether floating point arithmetic is to be emulated.

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.

Start target Endian

Select the way in which the byte order of ARMulator models that have a CP15 coprocessor is determined:

- Select Debug Endian to instruct the model to use the byte order set by the Debug Endian button.
- Select Hardware Endian 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.

Memory Map File

Specify a memory map file, or that you want to use default settings.

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 FPA instruction set.

If, in the Memory Map File box, 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*.

If you set a nonzero simulated Clock Speed, then the clock speed used is the value that you enter. Values stored in 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 5-65. The AXD clock speed defaults to 0.00 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.

For ARMulator, you do not need 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 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 `SWI_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.

`$memstats` is handled slightly differently because it does need a defined clock frequency, so that ARMulator can calculate how many wait states are needed for the memory speed defined in an `armsd.map` file. If you specify a clock speed and an `armsd.map` file is present, then `$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.

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).

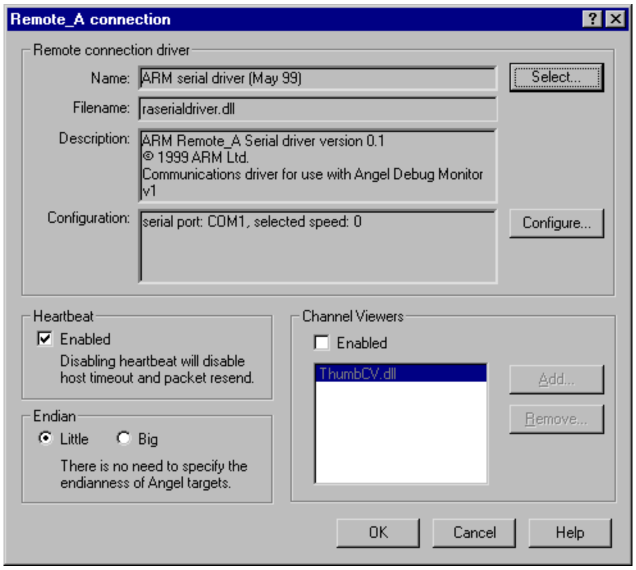
Some versions of Multi-ICE might also allow you to select a `.dll` file to use as a *debug communications channel* (DCC) viewer. Do not enable any DCC viewer from this dialog. Instead, use the AXD built-in viewer available from the Processor Views menu and enabled from the Processor Properties dialog. For more details see *Comms Channel processor view* on page 5-36 and *Configure Processor...* on page 5-91.

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 5-98.



Remote_A connection

Remote connection driver

Name: ARM serial driver (May 99)

Filename: raserialdriver.dll

Description: ARM Remote_A Serial driver version 0.1
© 1999 ARM Ltd.
Communications driver for use with Angel Debug Monitor v1

Configuration: serial port: COM1, selected speed: 0

Heartbeat
[✓ Enabled]
Disabling heartbeat will disable host timeout and packet resend.

Endian
[● Little] Big
There is no need to specify the endianness of Angel targets.

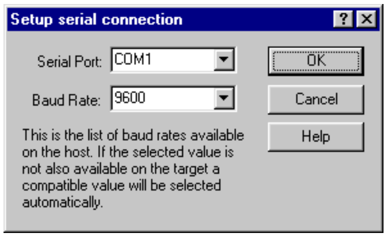
Channel Viewers
Enabled
ThumbCV.dll

Figure 5-98 Configuration of Remote_A connection

The Remote_A connection dialog allows 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 5-99, Figure 5-100, or Figure 5-101.

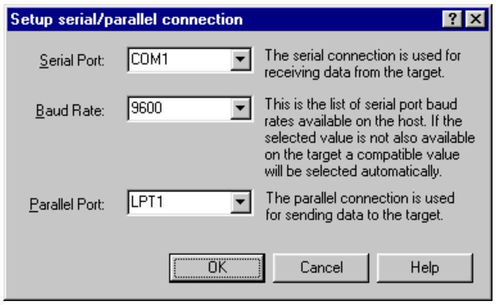

Setup serial connection

Serial Port: COM1

Baud Rate: 9600

This is the list of baud rates available on the host. If the selected value is not also available on the target a compatible value will be selected automatically.

Figure 5-99 Serial connection configuration


Setup serial/parallel connection

Serial Port: COM1

The serial connection is used for receiving data from the target.

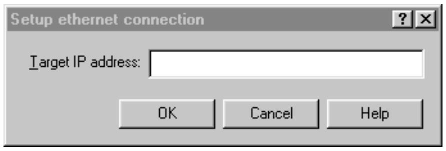
Baud Rate: 9600

This is the list of serial port baud rates available on the host. If the selected value is not also available on the target a compatible value will be selected automatically.

Parallel Port: LPT1

The parallel connection is used for sending data to the target.

Figure 5-100 Serial/parallel connection configuration


Setup ethernet connection

Target IP address:

Figure 5-101 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®, the **Endian** button sets the endianness of the target processor.

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 allows 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 5-36 and *Configure Processor...* on page 5-91.

Gateway configuration

If you need to add Gateway to the list of available targets in the Choose Target dialog, click **Add** and use the resulting browse dialog to locate and select the `gateway.dll` file.

Select the Gateway target line and click the **Configure** button to display the dialog shown in Figure 5-102. The Gateway Configuration dialog has at least two tabbed pages:

- Connection Details
- Advanced.

A third tabbed page is available if you use a Trace-enabled debugger. For details of this extra page, click on its **Help** button to display online help.

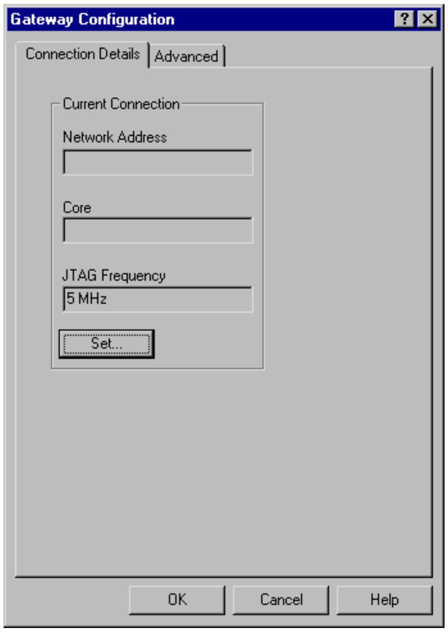


Figure 5-102 Gateway Configuration, Connection Details tab

On the Connection Details tabbed page, click **Set** to set connection details for your probe. The resulting Set Connection Details dialog is shown in Figure 5-103.

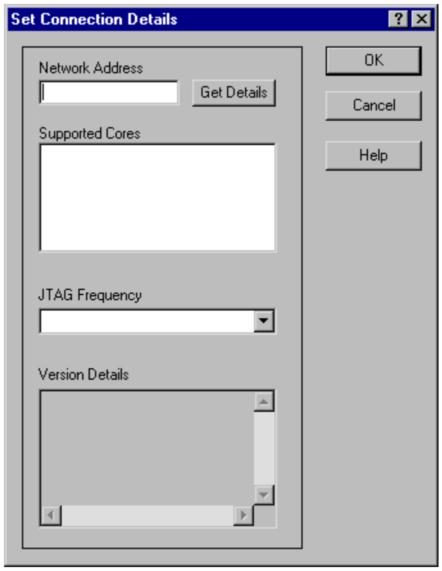


Figure 5-103 Gateway Configuration, Set Connection Details dialog

1. Enter the IP address for your probe in the IP Address field. See your HP documentation for more information on setting the IP address.

2. Click on the Lookup button. The Debugger establishes a connection with probe and displays a list of supported targets in the Supported Cores field.

3. Select the target type you want in the Supported Cores field.

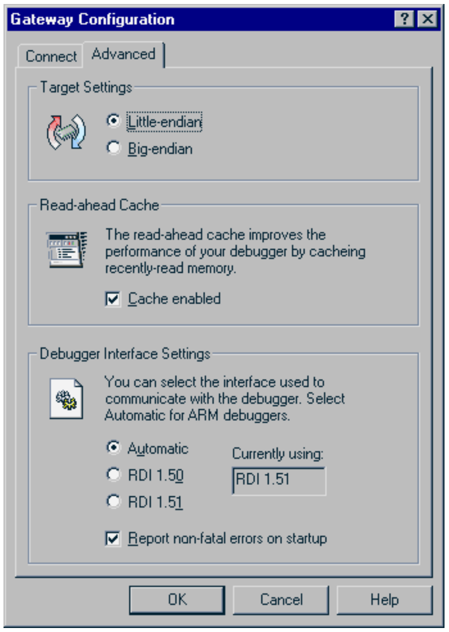
4. Select the JTAG base clock speed you require from the JTAG Frequency drop-down menu. This sets the frequency at which the probe clocks data across the target JTAG port. Higher frequencies give improved performance, especially for JTAG-intensive operations such as downloading.

   You are recommended to select the highest frequency supported by your target hardware. Hardware constraints such as stacking several devices together, or using long cables between the probe and the target, might require you to use lower JTAG frequencies.

5. Click **OK** to confirm your settings and close the Set Connection Details panel.

The Advanced tabbed page of the Gateway Configuration dialog, shown in Figure 5-104, contains three areas:

- Target Settings
- Read-ahead Cache
- Debugger Interface Settings.



Target Settings

[✓ Little-endian] Big-endian

Read-ahead Cache

The read-ahead cache improves the performance of your debugger by cacheing recently-read memory.

[✓ Cache enabled]

Debugger Interface Settings

You can select the interface used to communicate with the debugger. Select Automatic for ARM debuggers.

[✓ Automatic] RDI 1.50 RDI 1.51

Currently using: RDI 1.51

[✓ Report non-fatal errors on startup]

Figure 5-104 Gateway Configuration, Advanced tab

Normally these settings are correct, but you can change them if required.

The Target Settings area shows the byte order of the specified target:

Little-endian	Denotes a target that has the least significant byte of each word at the lowest memory address.
Big-endian	Denotes a target that has the most significant byte of each word at the lowest memory address.

The Read-ahead Cache option is enabled by default. Read-ahead cacheing improves memory read performance by reading more memory than requested by the debugger. The additional memory is cached in case it is needed later.

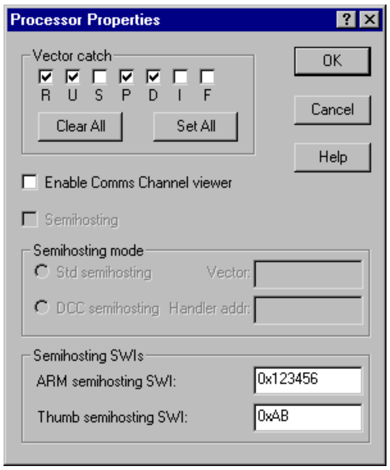
If you are debugging a system with demand-paged memory, deselect the Cache enabled option.

The radio buttons in the Debugger Interface Settings area allow you to specify the version of Remote Debugging Interface (RDI) you want Gateway to use for communicating with the debugger. Automatic is the default setting. Your debugger and Gateway determine the correct standard automatically. You can use this setting for all the ARM debuggers. If you need to specify a particular version of RDI, select either RDI 1.50 or RDI 1.51. The RDI version in use is displayed in Currently using.

By default, Gateway reports warnings as you connect to the target. If non-fatal error reports prevent you from starting a debugging session, you can stop them being reported by deselecting the Report non-fatal errors on startup check box.

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 tabbed page of the Control system view and selecting **Properties** from the resulting pop-up menu. A typical Processor Properties dialog is shown in Figure 5-105.


Processor Properties

Vector catch: R [✓] U [✓] S [ ] P [✓] D [✓] I [ ] F [ ]

Enable Comms Channel viewer [ ]

Semihosting [ ]

Semihosting mode: Std semihosting, Vector: ; DCC semihosting, Handler addr:

Semihosting SWIs

ARM semihosting SWI: 0x123456

Thumb semihosting SWI: 0xAB

Figure 5-105 Processor Properties dialog

The **Vector Catch** box allows 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 box 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 6-43.

The Enable Comms Channel viewer 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 Comms Channel). If you select the DCC semihosting mode, then the Comms Channel check box becomes disabled because the options are mutually exclusive.

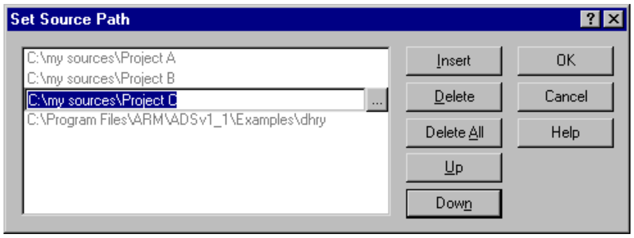
The Vector field sets the value of the `$semihosting_vector` variable. See the Semihosting chapter of the *ADS Debug Target Guide* for an explanation of this variable, and for more general information on semihosting issues.

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.

5.7.5 Source Path...

Select **Source Path...** from the Options menu to display the Set Source Path dialog shown in Figure 5-106. This specifies the paths that are searched, and the order in which they are searched, when a source file is required.



C:\my sources\Project A
C:\my sources\Project B
C:\my sources\Project C
C:\Program Files\ARM\ADSv1_1\Examples\dhry

Figure 5-106 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:\my sources\project B` 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 6-48, and *setsourcedir* on page 6-45.

5.7.6 Status Bar

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* 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

Point to **Profiling** to display a submenu, shown in Figure 5-107. This allows you to control profiling, provided you made suitable settings when you loaded the image. See *Profiling* on page 4-25 for details.

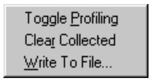


Figure 5-107 Profiling submenu

5.8 Window menu

The Window menu, shown in Figure 5-108, allows you to control the display of windows and icons on your screen.

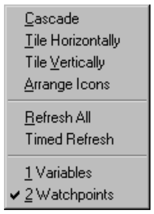


Figure 5-108 Window menu

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 5-95
- *Tile Horizontally* on page 5-95
- *Tile Vertically* on page 5-95
- *Arrange Icons* on page 5-95
- *Refresh All* on page 5-95
- *Timed Refresh* on page 5-96
- *List of relevant windows* on page 5-96.

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 only when you are debugging a target that supports RealMonitor (see *RealMonitor support* on page 4-13). 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 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.

To change the refresh interval, select **Options** → **Configure Interface** → **Timed Refresh** and set Refresh Interval (seconds) to a new value.

Timed Refresh is useful only when you are debugging a target that supports RealMonitor (see *RealMonitor support* on page 4-13). 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.

5.9 Help menu

The Help menu, shown in Figure 5-109, provides you with access to AXD online help and to details of the version of AXD that you are running.

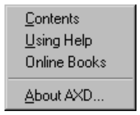


Figure 5-109 Help menu

The Help menu items and relevant toolbar icons are described under the following headings:

- *Contents* on page 5-97
- *Using help* on page 5-97
- *Online Books* on page 5-97
- *About AXD* on page 5-97
- *Toolbar icons* on page 5-98.

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 v1.1** → **Online Books**.

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.

Chapter 6
AXD Command-line Interface

This chapter describes the use of the *Command Line Interface* (CLI) window. It contains the following sections:

6.1 Command Line Window

Select **Command Line Interface** from the System Views menu to display the *Command Line Window* (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* on page 6-2
- *To automate repetitive tasks* on page 6-2.

To display the CLI window pop-up menu, right-click in the CLI window.

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 5-62 or *record* on page 6-36). You can then easily repeat the same commands by submitting the file to the CLI using the obey command (see *obey* on page 6-34).

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 5-62.

To display the CLI Properties dialog, shown in Figure 5-81 on page 5-63, Figure 5-82 on page 5-64, and Figure 5-83 on page 5-64, select **Properties...** from the pop-up menu.

The CLI Properties dialog allows 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 5-62 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+PageUp 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 5-62.

6.2 Parameters and prefixes

When entering commands, you might need 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
```

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 number 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.

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 need 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* on page 6-6
- *asm* on page 6-6
- *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 6-29.

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 4-15.

6.4.2 asm

The `asm` parameter must be set to `ARM`, `Thumb`, or `auto`.

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 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.

6.4.7 toggle

The *`toggle`* parameter must be set to `on` or `off`.

This parameter switches the associated command on or off.

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) or Thumb (16-bit). You must specify `ARM`, `Thumb`, 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
- 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 5-47 and *setprocprop* on page 6-43.
	`comms_channel`
	Enables or disables the communications channel.
	`semihosting_enabled`
	Enables or disables semihosting. Applies to Multi-ICE only. Can take one of the following values:
	`0` semihosting disabled
	`1` standard semihosting enabled
	`2` DCC semihosting enabled.
	`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`
  - `CoProc` *`n`*
- 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.

`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 `#`.

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 need.

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 6-45 and *sourcedir* on page 6-48.

6.6.2 backtrace

See *stackentries* on page 6-48.

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`.

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 6-40.

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 6-40) or select **Breakpoints...** from the System Views menu.

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:

```
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.

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 6-49 for a description of how to add a new reference point or display all reference points.

6.6.10 clearwatch

The `clearwatch` command unsets and deletes a specified watchpoint or all current watchpoints. See *watchpt* on page 6-53 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
```

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 6-48 for further information on stack entries.

This command does not change the execution context. It allows 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.

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 6-48).
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.

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`

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`, 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`.

6.6.17 echo

The `echo` command allows 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 checkbox 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 6-33.

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).

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`").

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`").

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 6-28 and *listformat* on page 6-29.

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 6-29).
`RDIName`	As shown in the last displayed format list.
`ShortName`	As shown in the last displayed format list, or on the Format tabbed page 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.

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 6-30.

6.6.26 go

See *run* on page 6-38.

6.6.27 help

The `help` command invokes AXD online help.

The shorthand form of the `help` command is `hlp`.

Syntax

```
hlp
```

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 6-37.

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 6-41 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]
```

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 6-29), are added to the list of available formats. A parameter allows 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.

The shorthand form of the `importformat` command is `impfmt`.

See also *format* on page 6-26.

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 6-45.

6.6.32 list

See *disassemble* on page 6-21.

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 6-26.

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()`.

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.

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 6-39.)

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 6-36.

There is no shorthand form of the `log` command.

Syntax

```
log[ file]
```

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).
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 6-53.

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 6-51 and *run* on page 6-38.

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 6-39.

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 6-31.

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 6-32.

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]
```

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 allows you to specify individual registers in other commands. See *setreg* on page 6-44, 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 6-36 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.

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 6-27 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).

6.6.54 runmode

No longer a valid command. See *stepsize* on page 6-50.

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
```

The `rto` causes execution to run to position 130 in the file specified by index 1 in the file list.

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.

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 6-31.)

The shorthand form of the `savesession` command is `ss`.

Syntax

```
ss file
```

where:

file	Specifies the session file to be created.

6.6.58 setbreakprops

The `setbreakprops` command allows 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
expression	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.

6.6.59 setimgprop

The `setimgprop` command sets an image-related internal variable to the specified value (see *imgproperties* on page 6-28). 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.

6.6.60 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.
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`.

6.6.61 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
```

6.6.62 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 `setprocessor` command is `sproc`.

Syntax

```
sproc processor
```

6.6.63 setprocprop

The `setprocprop` command sets a processor-related internal variable to the specified value (see *procproperties* on page 6-35). You have to supply either a string or an expression, depending on the type of the variable.

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

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 5-91.

```
spp vector_catch IuS
```

Sets the I and S bits on, and the U bit off, leaving the other bits unchanged.

`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*.

6.6.64 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 6-36). 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`.

6.6.65 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 6-48.

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 withing 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.

6.6.66 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 6-53).

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`.

6.6.67 setwatchprops

The `setwatchprops` command allows 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

Property name	Type
state	Flag (enable or disable)
processor	Processor
expression	String or value
log_text	String
value	Value
break_size	ASM
count	Integer

	Specify the name exactly as shown in the table, using lower-case 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.

`swp #2 value 17`

Set the value that is watched for by watchpoint number 2 to 17.

6.6.68 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.

6.6.69 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 6-45.

The shorthand form of the `sourcedir` command is `sdir`.

Syntax

```
sdir
```

6.6.70 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 allows you to specify individual stack entries in other commands. See *convariables* on page 6-20 and *context* on page 6-19, 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.

6.6.71 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
```

6.6.72 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
```

6.6.73 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.

6.6.74 step

The `step` command causes execution to proceed by one step.

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 6-50). If a subroutine call is encountered, this command steps over it.

6.6.75 stepsize

The `stepsize` command allows 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.

6.6.76 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]
```

6.6.77 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 installed.

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`

6.6.78 traceload

The `traceload` command loads a Trace configuration file. This command is effective only when the add-on product *Trace Debug Tools* (TDT) is installed.

The shorthand form of the `traceload` command is `trload`.

Syntax

```
trload tcfile
```

where:

tcfile	Specifies the Trace configuration file to be read.

6.6.79 type

See *source* on page 6-47.

6.6.80 unbreak

See *clearbreak* on page 6-17.

6.6.81 update

The `update` command allows 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 5-63.

6.6.82 unwatch

See *clearwatch* on page 6-19.

6.6.83 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]
```

6.6.84 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 6-45.

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.

6.6.85 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.

`Watching`	This describes what you are watching.
`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.
`Size`	This shows whether the watchpoint is ARM-sized (`4bytes`), Thumb-sized (`2bytes`), or a single byte (`1byte`).
`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 6-46.

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:

```
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.

6.6.86 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 6-48).

There is no shorthand form of the `where` command.

Syntax

```
where[ context]
```

Part B

ADW and ADU

Chapter 7
About ADW and ADU

This chapter introduces *ARM Debugger for Windows* (ADW) and *ARM Debugger for UNIX* (ADU). These are two versions of the same debugger, adapted to run under Windows and UNIX respectively. It contains the following sections:

7.1 About the ADW and ADU debuggers

ADW formed part of the *ARM Software Development Toolkit* (SDT), and is also supplied with the *ARM Developer Suite* (ADS). ADU was an extra-cost addition in SDT 2.11a or greater, and is also included in ADS.

ADW and ADU enable you to run and debug your ARM-targeted image using any of the debugging systems described in *Debugging systems* on page 7-5.

You can also use ADW and ADU to benchmark your application.

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

Note

ADW and ADU screens differ slightly in appearance. Your screens, therefore, might look different from the figures in this part of the book.

In the past the ARM C++ compiler was an extra-cost option, and its installation added extra features to ADW and ADU to support debugging C++. The C++ compiler is supplied as a standard part of ADS, so making the extra features available in all cases. Refer to Chapter 10 *Using ADW and ADU with C++* for details.

Most of Part B of this book applies to both ADW and ADU. If a section applies to one of those debuggers only, this is indicated in the text or in the section heading.

7.2 Online help

When you have started ADW or ADU, you can display online help giving information about your current situation, or navigate your way to any other page of ADW and ADU online help as follows:

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

Help button Many ADW and ADU windows contain a **Help** button. Click this button to display help on the currently active window.

Help menu Select **Contents** from the Help menu to display a Help Topics screen with Contents, Index, and Find tabs. The tab you used last is selected. Click either of the other tabs to change the selection.

Select **Search** from the Help menu to display the Help Topics screen with the Index tab selected.

On the Contents tabbed page, click on a closed book to open it and see a list of the topics it contains. Select a topic and click the **Display** button to display online help. Click on an open book to close it.

On the Index tabbed page, either scroll through the list of entries or start typing an entry to bring into view the index entry you want. Select an index entry and click the **Display** button to display online help.

On the Find tabbed page, follow the instructions to search the online help text for any keywords you specify. The first time you use Find a database file is constructed, and is then available for any later Find operations.

Select **Using Help** from the Help menu to display a guide to on-screen help.

Hypertext links

Most pages of online help include highlighted text you can click on to display other relevant online help:

- highlighted text underscored with a broken line displays a pop-up box
- highlighted text underscored with a solid line jumps to another page of help.

Browse buttons

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

7.3 Debugging an ARM application

ADW and ADU work in conjunction with either:

- a hardware target system, such as an ARM Development Board, communicating through Multi-ICE, EmbeddedICE, or Angel
- a software target system, such as ARMulator.

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

To debug your application you must choose:

- a *debugging system* that can be:
  - — hardware-based on an ARM core (Multi-ICE, for example)
  - — software that simulates an ARM core (ARMulator, for example)
  - — software that runs on a target (Angel, for example).
- a *debugger,* for example ADW, ADU, or armsd.

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

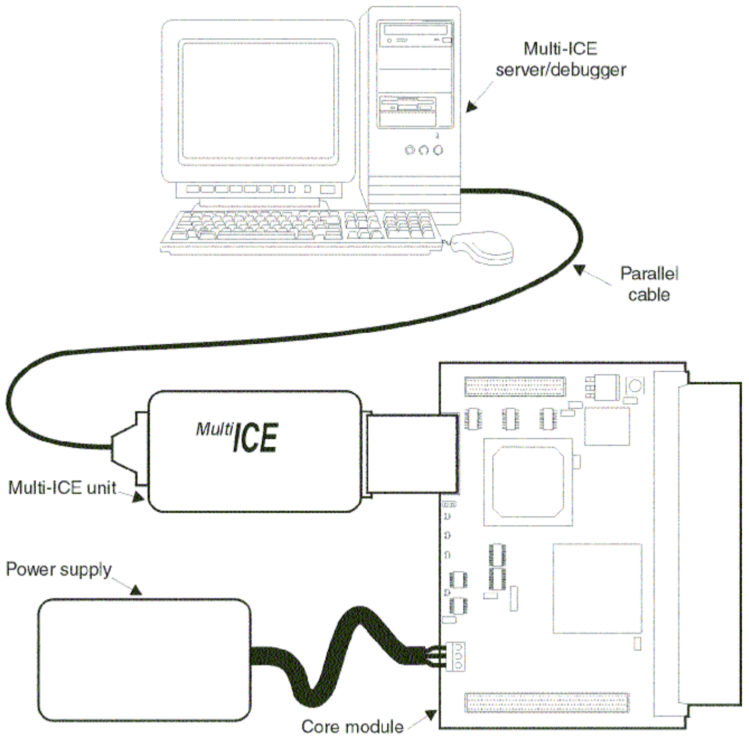


Figure 7-1 A typical debugging set-up

7.4 Debugging systems

The following debugging systems are available for applications developed to run on an ARM core:

- *ARMulator* on page 7-5
- *Multi-ICE and EmbeddedICE* on page 7-5
- *Angel debug monitor* on page 7-6
- *Gateway* on page 7-6.

7.4.1 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 have taken. As a result, ARMulator is well suited to software development and benchmarking.

7.4.2 Multi-ICE and EmbeddedICE

Multi-ICE and EmbeddedICE are JTAG-based debugging systems for ARM processors. Multi-ICE and EmbeddedICE provide the interface between a debugger and an ARM core embedded within an ASIC. 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).

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

For information on configuration options see *Target Configuration* on page 9-33. For detailed information on Multi-ICE refer to the Multi-ICE documentation.

The debugger internal variable $top_of_memory has a default value of 0x80000. This is the required value when you are using an ARM PID board as the target. An 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 window* on page 8-15.

7.4.3 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.

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.

7.4.4 Gateway

Gateway provides facilities to enable ADW and ADU 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 CPU and coprocessor registers. ADW or ADU accesses these facilities across an Ethernet connection. For further information on Agilent emulators and similar products see *Third party products* on page xi.

Gateway handles all aspects of setting up and maintaining the Ethernet connection with the emulator. However, you must provide the correct IP address and specify the core to be debugged using the Gateway configuration dialog (see *Gateway configuration* on page 9-42). It is also your responsibility to ensure that the emulation probe and TIM are suitable for the application core.

Note

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

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

7.5 Debugger concepts

This section introduces concepts involved in debugging program images, including:

- *Debug agent* on page 7-7
- *Remote debug interface* on page 7-7.

7.5.1 Debug agent

A debug agent is the entity that performs the actions requested by the debugger, such as setting breakpoints, reading from memory, or writing to memory. It is not the program being debugged, or the ARM Debugger itself. Examples of debug agents include Multi-ICE, EmbeddedICE, ARMulator, and Angel Debug Monitor.

7.5.2 Remote debug interface

The *Remote Debug Interface* (RDI) is an open ARM standard procedural interface between a debugger and the debug agent. The widest possible adoption of this standard is encouraged.

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).

Chapter 8
Getting Started in ADW and ADU

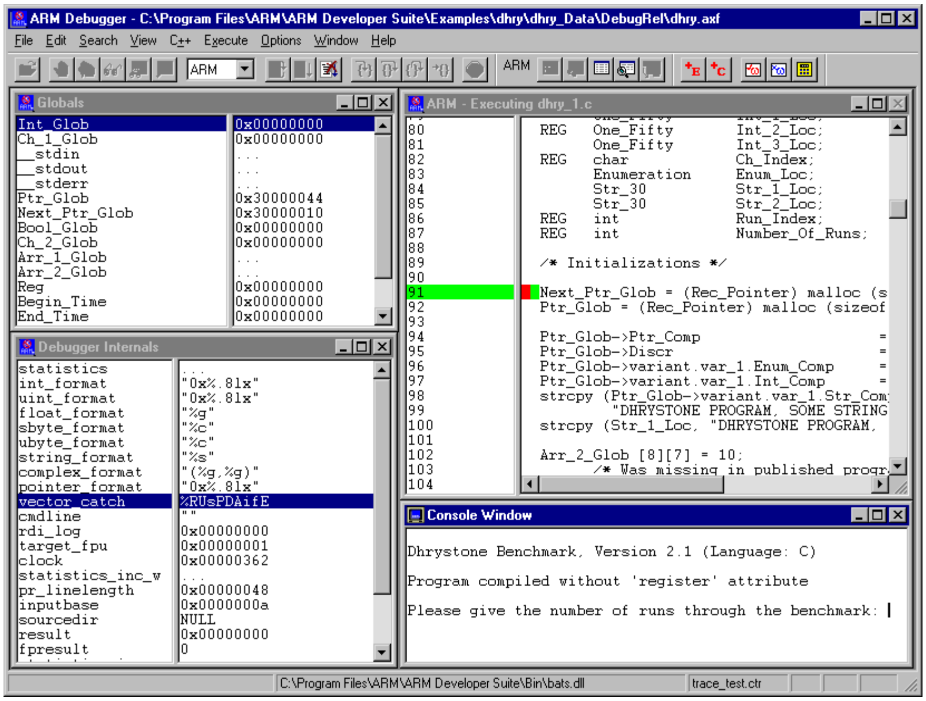
This chapter describes the main features of the ADW and ADU desktop and gives you enough information to start working with the debugger. Additional features are described in Chapter 9 *Working with ADW and ADU*. This chapter contains the following sections:

8.1 The ADW and ADU desktop

The main features of the ADW and ADU desktop are:

- A menu bar, toolbar, mini toolbar, and status bar. For details see *Menu bar, toolbar, mini toolbar, and status bar* on page 8-3.
- A number of windows displaying a variety of information as you debug your executable image. For details see *ADW and ADU desktop windows* on page 8-11.
- A window-specific menu that is available for each window, as described in *ADW and ADU desktop windows* on page 8-11.

Figure 8-1 shows ADW or ADU with the Execution, Console, Globals, and Debugger Internals windows, in the process of debugging the sample image DHRY.


ARM Debugger - C:\Program Files\ARM\ARM Developer Suite\Examples\dhry\dhry_Data\DebugRel\dhry.axf
File Edit Search View C++ Execute Options Window Help
ARM
ARM
Globals
Debugger Internals
ARM - Executing dhry_1.c
Console Window
C:\Program Files\ARM\ARM Developer Suite\Bin\bats.dll
trace_test.ctr

Int_Glob	0x00000000
Ch_1_Glob	0x00000000
__stdin	...
__stdout	...
__stderr	...
Ptr_Glob	0x30000044
Next_Ptr_Glob	0x30000010
Bool_Glob	0x00000000
Ch_2_Glob	0x00000000
Arr_1_Glob	...
Arr_2_Glob	...
Reg	0x00000000
Begin_Time	0x00000000
End_Time	0x00000000

statistics	...
int_format	"0x%.8lx"
uint_format	"0x%.8lx"
float_format	"%g"
sbyte_format	"%c"
ubyte_format	"%c"
string_format	"%s"
complex_format	"(%g,%g)"
pointer_format	"0x%.8lx"
vector_catch	%RUsPDAifE
cmdline	""
rdi_log	0x00000000
target_fpu	0x00000001
clock	0x00000362
statistics_inc_w	...
pr_linelength	0x00000048
inputbase	0x0000000a
sourcedir	NULL
result	0x00000000
fpresult	0

```
        REG   One_Fifty        Int_2_Loc;
              One_Fifty        Int_3_Loc;
        REG   char             Ch_Index;
              Enumeration      Enum_Loc;
              Str_30           Str_1_Loc;
              Str_30           Str_2_Loc;
        REG   int              Run_Index;
        REG   int              Number_Of_Runs;

  /* Initializations */

  Next_Ptr_Glob = (Rec_Pointer) malloc (s
  Ptr_Glob = (Rec_Pointer) malloc (sizeof

  Ptr_Glob->Ptr_Comp                    =
  Ptr_Glob->Discr                       =
  Ptr_Glob->variant.var_1.Enum_Comp     =
  Ptr_Glob->variant.var_1.Int_Comp      =
  strcpy (Ptr_Glob->variant.var_1.Str_Com
          "DHRYSTONE PROGRAM, SOME STRING
  strcpy (Str_1_Loc, "DHRYSTONE PROGRAM,

  Arr_2_Glob [8][7] = 10;
          /* Was missing in published progr
```

```
Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without 'register' attribute

Please give the number of runs through the benchmark:
```

Figure 8-1 A typical ADW or ADU desktop display

8.1.1 Menu bar, toolbar, mini toolbar, and status bar

The menu bar is at the top of the ADW and ADU desktop. Click on a menu name to display the pull-down menu.

Underneath the menu bar is the toolbar. Position the cursor over an icon and a brief description is displayed. A processor-specific mini toolbar is also displayed. The menus, the toolbar, and the mini toolbar are described in greater detail in the online help.

At the bottom of the desktop is the status bar. This provides current status information or describes the currently selected user interface component.

8.2 Starting and closing ADW and ADU

Starting and closing ADW and ADU are described in the subsections:

- *Starting ADW* on page 8-4
- *Starting ADU* on page 8-4
- *ADW and ADU arguments* on page 8-5
- *Closing ADW and ADU* on page 8-6.

8.2.1 Starting ADW

Start ADW in any of the following ways:

- if you are running Windows 95 or Windows 98, click on the **ADW Debugger** icon in the ADSv1_1 program folder or select **Start** → **Programs** → **ARM Developer Suite v1.1** → **ADW Debugger**
- if you are running Windows NT4, double-click on the **adw.exe** icon in the ADSv1_1\Bin Program group or select **Start** → **Programs** → **ARM Developer Suite v1.1** → **ADW Debugger**
- if you are working in the CodeWarrior IDE, open a project and select **Edit** → ***target*** **Settings...** → **Debugger** → **ARM Debugger** to ensure that ADW is the default debugger, select **ARM Runner** to ensure that ADW is the default runner, make any other settings, **Save** the settings, then click **Run/Debug** or select **Debug** from the Project menu
- launch ADW from the DOS command line, optionally with arguments.

8.2.2 Starting ADU

Start ADU in either of the following ways:

- from any directory type the full path and name of the debugger, for example, `/opt/arm/adu`
- change to the directory containing the debugger and type its name, for example, `./adu`

8.2.3 ADW and ADU arguments

The possible arguments (which must be in lower case) for both ADW and ADU are:

`-debug` *`ImageName`*
: Load *`ImageName`* for debugging.

`-exec` *`ImageName`*
: Load and run *`ImageName`*.

`-reset` Reset the registry settings to defaults.

`-nologo` Do not display the splash screen on startup.

`-nowarn` Do not display the warning when starting remote debugging.

`-nomainbreak` Do not set a breakpoint on `main()` on loading image.

`-script` *`ScriptName`*
: Obey the *`ScriptName`* on startup. This is the equivalent of typing `obey` *`ScriptName`* as soon as the debugger starts up.

`-symbols` Load only the symbols of the specified image. This is equivalent to selecting **Load Symbols only…** from the File menu.

`-li, -bi` Start the debugger in little-endian or big-endian mode.

`-args` Pass the remaining command-line arguments to the specified image.

`-armul` Start the debugger using ARMulator.

`-adp` `-linespeed` *`baudrate`* `[-port [s=`*`serial port`*`[,p=`*`parallel port`*`]] | [e=`*`ethernet address`*`]]`

Start the debugger using Remote_A, if available in the current RDI connection list.

You can use `-linespeed` *`baudrate`* only in conjunction with `-adp`, to specify the baud rate of the connection.

You can use `-port` only in conjunction with `-adp`, to specify the connection to the device.

`-session` *`SessionName`*

Use this field to specify an ADW session name (which must contain no space characters). You can use this option to save ADW configuration settings in the Windows registry:

- if you specify a new session name, ADW creates a new named session and saves the configuration information for the current debug session in the Windows registry when you exit ADW
- if you specify a previously used session name, ADW is configured using the information in the named session.

This option is useful for saving and restoring multiple configurations for use with Multi-ICE, or in any other case where you want to restore your previous ADW configuration.

As an example of the use of arguments, to launch ADW from the command line and load `sorts.axf` for debugging, but without setting a breakpoint on `main()`, type:

```
adw -debug sorts.axf -nomainbreak
```

To launch ADW (with arguments) from the CodeWarrior IDE, select **Target Settings...** → **Debugger** → **ARM Debugger**, select ADW and specify any arguments you want to be supplied to the debugger.

Refer to *Specifying command-line arguments for your program* on page 9-22 for more information on specifying command-line options.

8.2.4 Closing ADW and ADU

Select **Exit** from the File menu to close down ADW or ADU.

8.3 Loading, executing, and reloading a program image

You must load a program image before you can execute it or step through it.

8.3.1 Loading an image

To load a program image:

1. Select **Load Image** from the File menu or click the **Open File** button. The Open File dialog is displayed.
2. Select the filename of the executable image you want to debug.
3. Enter in the Arguments box any command-line arguments your image needs.
4. Click **OK**. The program is displayed in the Execution window as disassembled code.

   A breakpoint is automatically set at the entry point of the image, usually the first line of source after the `main()` function. The current execution marker, a green bar indicating the current line, is located at the entry point of the program.

If you have recently loaded your required image, your file appears as a recently used file on the File menu. If you load your image from the recently used file list, ADW or ADU loads the image using the command-line arguments you specified in the previous run.

If the image you are loading uses floating point data, the `$target_fpu` debugger internal variable must match the image. See *Debugger Internals window* on page 8-15 and Table 12-2 on page 12-7.

8.3.2 Executing an image

To run your program in ADW or ADU, select **Go** from the Execute menu or click the **Go** button to execute the entire program. Execution continues until:

- a breakpoint halts the program at a specified point
- a watchpoint halts the program when a specified variable or register changes
- you stop the program by clicking the **Stop** button.

Alternatively, select **Step** from the Execute menu or click the **Step** button to step through the code a line at a time (see *Stepping through an image* on page 9-11).

While the program executes:

- the Console window is active, provided semihosting is in operation (see *ADS Debug Target Guide* for more information)
- the program code is displayed in the Execution window.

To continue execution from the point where the program stopped use **Go** or **Step**.

Note

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

8.3.3 Reloading an image

To reload an executable image, select **Reload Current image** from the File menu or click the **Reload** button on the toolbar.

8.4 Examining and setting variables, registers, and memory

You can use ADW or ADU to display and modify the contents of the variables and registers used by your executable image. You can also examine the contents of memory.

8.4.1 Variables

To display or modify local or global variables:

1. Display either the Locals or Globals window:
   a. Select **View** → **Variables** → **Local** or click the **Locals** button on the toolbar to display a list of local variables.
   b. Select **View** → **Variables** → **Global** to display a list of global variables.
2. Double-click on the value you want to change in the right pane of the window. Generally, in-place editing is possible allowing you to change the selected value. If necessary, a Memory window is displayed or the variable is expanded or accessed indirectly.
3. Press Return when you have set the variable to the required value, or click away from the value to cancel the editing.

8.4.2 Registers

To display or modify registers for the *current* processor mode, click the **Registers** button on the toolbar.

To display or modify registers for a *selected* processor mode:

1. Select the **Registers** submenu from the View menu.
2. Select the required processor mode from the Registers submenu. The registers are displayed in the appropriate Registers window.

To change the value held in a register, double-click on its current value in the right pane of its window.

Generally, in-place editing is possible, allowing you to change the selected value. Press Return when you have set the register to the required value, or click away from the value to cancel the editing.

If in-place editing is not possible, a dialog is displayed allowing you to edit the value stored in the register.

8.4.3 Memory

To display the contents of a particular area of memory:

1. Select **Memory** from the View menu or click on the **Memory** button. The Memory Address dialog is displayed.
2. Enter the address as a hexadecimal value (prefixed by `0x`) or as a decimal value. You can also enter an expression, for example `@main + 0x5C`.
3. Click **OK**. The Memory window opens and displays the contents of memory around the address you specified.

When you have opened the Memory window you can:

- display other parts of the current 4KB area of memory by using the scrollbar
- display more remote areas of memory by entering another address
- right-click anywhere in the window to display the Memory window menu, allowing you to display the contents as words, halfwords, or bytes with ASCII characters.

To enter another address range:

1. Select **Goto** from the Search menu or select **Goto address** from the Memory Window menu. The Goto Address dialog is displayed.
2. Enter an address as a hexadecimal value (prefixed by `0x`) or as a decimal value. You can also enter an expression, for example `@main + 0x5C`.
3. Click **OK**.

See *Saving an area of memory to disk* on page 9-20 for more information on working with areas of memory.

8.5 ADW and ADU desktop windows

The first time you run ADW or ADU, you see the:

- *Execution window* on page 8-12
- *Console window* on page 8-13
- *Command window* on page 8-14.

The can also use the View menu to display the:

- *Backtrace window* on page 8-14
- *Breakpoints window* on page 8-15
- *Debugger Internals window* on page 8-15
- *Disassembly window* on page 8-15
- *Expression window* on page 8-16 (from the Variables submenu)
- *Function Names window* on page 8-16
- *Locals/Globals window* on page 8-16 (from the Variables submenu)
- *Low Level Symbols window* on page 8-17
- *Memory window* on page 8-17
- *RDI Log window* on page 8-18
- *Registers window* on page 8-18
- *Search Paths window* on page 8-18
- *Source File window* on page 8-19
- *Source Files List window* on page 8-19
- *Watchpoints window* on page 8-19.

Some windows become available only after you have loaded an image.

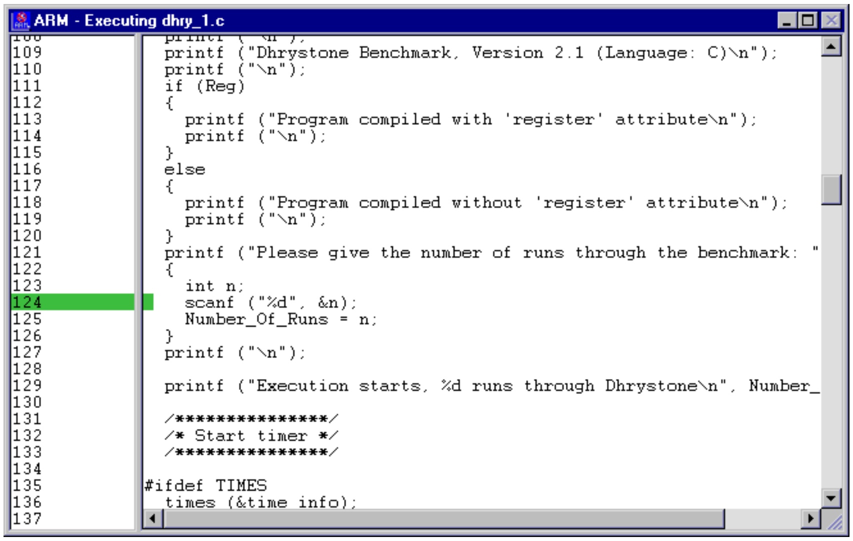
Each of the ADW and ADU desktop windows displays a window-specific menu when you click the secondary mouse button over the window. The secondary button is typically the right mouse button. To activate an item-specific option you must position the cursor over the item in the window before clicking.

Each of the window-specific menus is described in the online help for that window.

You can change the format of displayed windows, and the settings are automatically saved for future use. When you start the debugger you see the arrangement of windows you were using when you last quit ADW or ADU.

8.5.1 Execution window

The Execution window (see Figure 8-2) displays the source code of the currently executing program.



```
  printf ("Dhrystone Benchmark, Version 2.1 (Language: C)\n");
  printf ("\n");
  if (Reg)
  {
    printf ("Program compiled with 'register' attribute\n");
    printf ("\n");
  }
  else
  {
    printf ("Program compiled without 'register' attribute\n");
    printf ("\n");
  }
  printf ("Please give the number of runs through the benchmark: "
  {
    int n;
    scanf ("%d", &n);
    Number_Of_Runs = n;
  }
  printf ("\n");

  printf ("Execution starts, %d runs through Dhrystone\n", Number_

  /****************/
  /* Start timer */
  /****************/

#ifdef TIMES
  times (&time_info);
```

Figure 8-2 Execution window

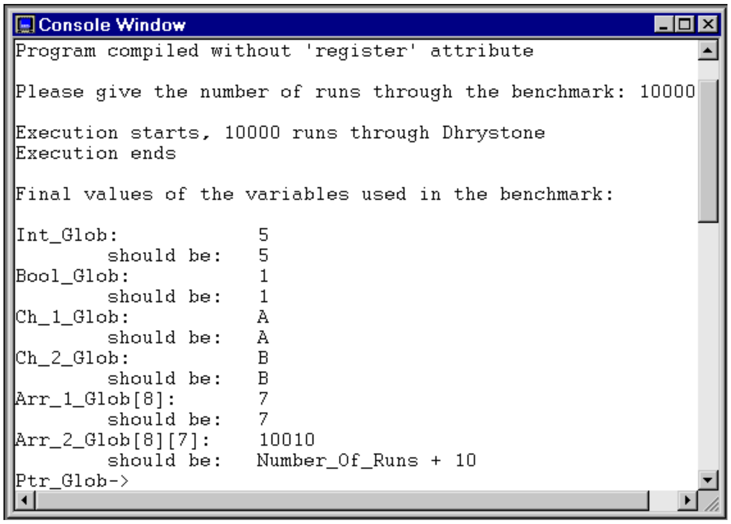
Use the Execution window to:

- execute the entire program or step through the program line by line
- change the display mode to show disassembled machine code interleaved with high-level C or C++ source code
- display another area of the code by address
- toggle, set, edit, or delete breakpoints.

8.5.2 Console window

The Console window (see Figure 8-3) allows you to interact with the executing program. Anything printed by the program, for example a prompt for user input, is displayed in this window and any input required by the program must be entered here.

Information remains in the window until you select **Clear** from the Console window menu. You can also save the contents of the Console window to disk, by selecting **Save** from the Console window menu.



```
Program compiled without 'register' attribute

Please give the number of runs through the benchmark: 10000

Execution starts, 10000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    10010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
```

Figure 8-3 Console window

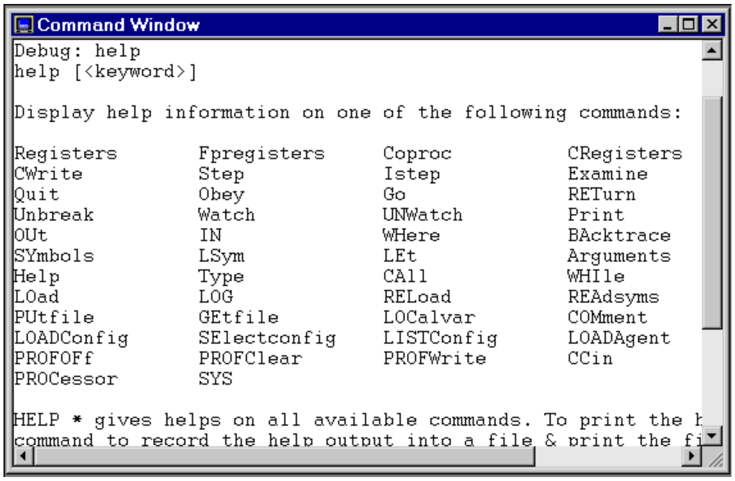
Initially the Console window displays the startup messages of your target processor, for example ARMulator or ARM Development board.

——— Note ———

When the executing image requires input from the debugger keyboard, most ADW and ADU functions are disabled until you have entered that information.

8.5.3 Command window

Use the Command window (see Figure 8-4) to enter armsd instructions when you are debugging an image.



```
Debug: help
help [<keyword>]

Display help information on one of the following commands:

Registers       Fpregisters     Coproc          CRegisters
CWrite          Step            Istep           Examine
Quit            Obey            Go              RETurn
Unbreak         Watch           UNWatch         Print
OUt             IN              WHere           BAcktrace
SYmbols         LSym            LEt             Arguments
Help            Type            CAll            WHIle
LOad            LOG             RELoad          READsyms
PUtfile         GEtfile         LOCalvar        COMment
LOADConfig      SElectconfig    LISTConfig      LOADAgent
PROFOFf         PROFClear       PROFWrite       CCin
PROCessor       SYS

HELP * gives helps on all available commands. To print the h
command to record the help output into a file & print the fi
```

Figure 8-4 Command window

See *Using command-line debugger instructions* on page 9-22 for further details about the use of the Command window. Type `help` at the Debug prompt for information on the available commands or refer to Part C of this book.

8.5.4 Backtrace window

The Backtrace window displays current backtrace information about your program. Use this to:

- show disassembled code for the current procedure
- show a list of local variables for the current procedure
- toggle, set, edit, or delete breakpoints.

8.5.5 Breakpoints window

The Breakpoints window displays a list of all breakpoints set in your image. The actual breakpoint is displayed in the right-hand pane. If the breakpoint is on a line of code, the relevant source file is shown in the left-hand pane.

Use the Breakpoints window to:

- show source or disassembled code
- edit or remove breakpoints.

To set a new breakpoint, see *Source File window* on page 8-19.

8.5.6 Debugger Internals window

The Debugger Internals window displays some of the internal variables used by ADW and ADU. These internal variables are also used by armsd, and details are given in *armsd variables* on page 12-7.

You can use the Debugger Internals window to examine the values of these variables, and to change the values of all except those marked read-only in the table. For information about display formats see *Working with variables* on page 9-13.

8.5.7 Disassembly window

The Disassembly window displays disassembled code interpreted from a specified area of memory. Memory addresses are listed in the left-hand pane and disassembled code is displayed in the right-hand pane. You can view ARM code, Thumb code, or both.

Use the Disassembly window to:

- go to another area of memory
- change the disassembly mode to ARM, Thumb, or Mixed
- set, edit, or remove breakpoints.

Note

More than one Disassembly window can be active at a time.

For details of displaying disassembled code, see *Displaying disassembled and interleaved code* on page 9-16.

8.5.8 Expression window

The Expression window displays the values of selected variables and registers.

Use the Expression window to:

- change the format of selected items, or all items
- edit or delete expressions
- display the section of memory pointed to by the contents of a variable.

For more information on displaying variable information, see *Working with variables* on page 9-13.

8.5.9 Function Names window

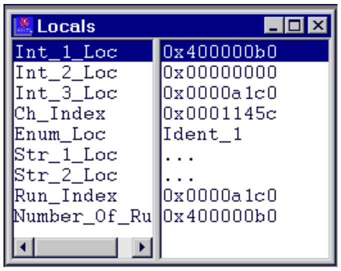
The Function Names window lists the functions that are part of your program.

Use the Function Names window to:

- display a selected function as source code
- set, edit, or remove a breakpoint on a function.

8.5.10 Locals/Globals window

The Locals window (see Figure 8-5) displays a list of variables currently in scope. The Globals window displays a list of global variables. The variable name is displayed in the left-hand pane, the value is displayed in the right-hand pane.



```
Int_1_Loc      0x400000b0
Int_2_Loc      0x00000000
Int_3_Loc      0x0000a1c0
Ch_Index       0x0001145c
Enum_Loc       Ident_1
Str_1_Loc      ...
Str_2_Loc      ...
Run_Index      0x0000a1c0
Number_Of_Ru   0x400000b0
```

Figure 8-5 Locals window

Use the Locals/Globals window to:

- change the content of a variable (double-click on the value)
- display the section of memory pointed to by a variable
- change the display format for the selected value, or for all values in the window
- set, edit, or remove a watchpoint on a variable
- double-click on an item to expand a structure (the details are displayed in another variable window).

As you step through the program, the variable values are updated.

For more information on displaying variable information, see *Working with variables* on page 9-13.

8.5.11 Low Level Symbols window

The Low Level Symbols window displays a list of all the low-level symbols in your program.

Use the Low Level Symbols window to:

- display the memory pointed to by the selected symbol
- display the source or disassembled code pointed to by the selected symbol
- set, edit, or remove a breakpoint on the line of code pointed to by the selected symbol.

You can display the low-level symbols in either name or address order. Right-click in the window to display the Low Level Symbols window menu and select **Sort Symbols by…** to toggle between the two settings.

8.5.12 Memory window

The Memory window displays the contents of an area of memory surrounding a specified address. Addresses are listed in the left-hand pane, and the memory content is displayed in the right-hand pane.

Use the Memory window to:

- display other areas of memory by scrolling or specifying an address
- set, edit, or remove a watchpoint
- change the contents of memory (double-click on an address)
- change the format of the display.

You can open multiple Memory windows.

8.5.13 RDI Log window

The RDI Log window displays the low-level communication messages between ADW or ADU and the target processor.

Note

This facility is not normally enabled (see *Remote debug information* on page 9-17).

8.5.14 Registers window

The Registers window displays the registers corresponding to the mode named at the top of the window, with the contents displayed in the right-hand pane. You can double-click on an item to modify the value in the register, unless you are debugging an Angel target when you can change the values in the registers of the current mode only.

Use the Registers window to:

- display the contents of the register memory
- display the memory pointed to by the selected register
- edit the contents of a register
- set, edit, or remove a watchpoint on a register.

You can, for example, double-click on the value of a program status register to change its settings.

Note

Multiple register mode windows can be open at any one time, but you cannot open more than one window for each processor mode. For example, you can open no more than one FIQ register window at a time.

8.5.15 Search Paths window

The Search Paths window displays the search paths of the image currently being debugged. You can remove a search path from this window using the Delete key.

8.5.16 Source File window

The Source File window displays the contents of the source file named at the top of the window. Line numbers are displayed in the left-hand pane, code in the right-hand pane.

Use the Source File window to:

- search for a line of code by line number
- set, edit, or remove breakpoints on a line of code
- toggle the interleaving of source and disassembly.

For more information on displaying source files, see *Working with source files* on page 9-12.

8.5.17 Source Files List window

The Source Files List window displays a list of all source files that have contributed debug information to the loaded image.

Use the Source Files List window to select a source file that is displayed in its own Source File window.

8.5.18 Watchpoints window

The Watchpoints window displays a list of all watchpoints.

Use the Watchpoints window to:

- delete a watchpoint
- edit a watchpoint.

To set a new watchpoint, see *Memory window* on page 8-17.

Chapter 9
Working with ADW and ADU

This chapter describes more of the features of ADW and ADU. It contains the following sections:

9.1 Breakpoints, watchpoints, backtracing, and stepping

You use breakpoints and watchpoints to stop program execution when a selected line of code is about to be executed, or when a specified condition occurs. You can also execute your program step by step. This section contains the following subsections:

- *Breakpoints* on page 9-2
- *Watchpoints* on page 9-7
- *Backtrace* on page 9-10
- *Stepping through an image* on page 9-11.

9.1.1 Breakpoints

A breakpoint is a point in the code where your program is halted by ADW or ADU. When you set a breakpoint it is marked in red in the left pane of the breakpoints window.

There are two types of breakpoint:

- a simple breakpoint that stops at a particular point in your code
- a complex breakpoint that:
  - — stops when the program has passed the specified point a number of times
  - — stops at the specified point only when an expression is true.

You can set a breakpoint at a point in the source, or in the disassembled code if it is currently being displayed. To display the disassembled code, either:

- select **Toggle Interleaving** from the Options menu to display interleaved source and assembly language in the Execution window
- select **Disassembly...** from the View menu to display the Disassembly window.

You can also set breakpoints on individual statements on a line, if that line contains more than one statement.

You can set, edit, or delete breakpoints in the following windows:

- Execution
- Disassembly
- Source File
- Backtrace
- Breakpoints
- Function Names
- Low Level Symbols
- Class View (applicable to C++ only).

Setting a simple breakpoint

There are two methods you can use to set a simple breakpoint:

- Method 1
    1. Double-click on the line where you want to set the breakpoint.
    2. Click the **OK** button in the dialog box that appears.
- Method 2
    1. Position the cursor in the line where you want to set the breakpoint.
    2. Set the breakpoint in any one of the following ways:
        - select **Toggle Breakpoint** from the Execute menu
        - click the **Toggle breakpoint** button
        - press the F9 key.

A new breakpoint is displayed as a red marker in the left pane of the Execution window, the Disassembly window, or the Source File window.

In a line with several statements you can set a breakpoint on an individual statement, as shown in the following example:

```
int main()
{
    hello(); world();
    .
    .
    .
    return 0;
}
```

If you position the cursor on the word `world` and click the **Toggle breakpoint** button, `hello()` is executed, and execution halts before `world()` is executed.

To see all the breakpoints set in your executable image select **Breakpoints** from the View menu.

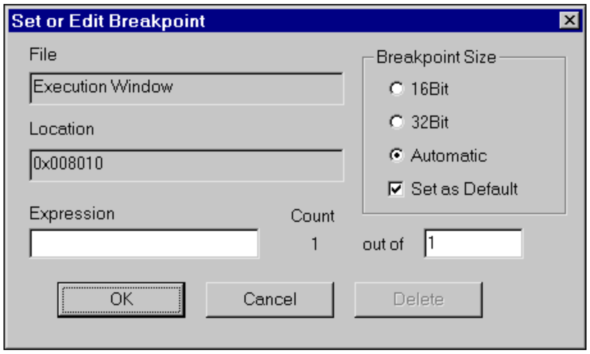
To set a simple breakpoint on a function:

1. Display a list of function names in the Function Names window by selecting **Function Names** from the View menu.
2. Select **Toggle Breakpoint** from the Function Names window menu or click the **Toggle breakpoint** button.

The breakpoint is set at the first statement of the function. In a Low Level Symbols window, the breakpoint is set to the first machine instruction of the function, that is, at the beginning of its entry sequence.

Complex breakpoints

When you set a complex breakpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Breakpoint dialog (see Figure 9-1).



File: Execution Window
Location: 0x008010
Expression:
Count: 1 out of 1
Breakpoint Size: ( ) 16Bit ( ) 32Bit [✓ Automatic] [✓ Set as Default]

Figure 9-1 Set or Edit Breakpoint dialog

This dialog contains the following fields:

File The source file that contains the breakpoint. This field is read-only.

Location The position of the breakpoint within the source file. This position is a hexadecimal address for assembler code. For C or C++ code, it is shown as a function name, followed by a line number, and if the line contains multiple statements, a column position. This field is read-only.

Expression An expression that must be true for the program to halt, in addition to any other breakpoint conditions. Use C-like operators such as:

```
i < 10
i != j
i != j + k
```

Count The program halts when all the breakpoint conditions apply for the *n*th time.

Breakpoint Size

You can set breakpoints to be 32-bit (ARM) or 16-bit (Thumb) size, or allow the debugger to make the appropriate setting. A checkbox allows you to make your selection the default setting.

Setting or editing a complex breakpoint

You can set complex breakpoints on:

- a line of code
- a function
- a low-level symbol.

To set or edit a complex breakpoint on a line of code:

1. Double-click on the line where you want to set a breakpoint, or on an existing breakpoint position. The Set or Edit Breakpoint dialog is displayed.
2. Enter or alter the details of the breakpoint.
3. Click **OK**. The breakpoint is displayed as a red marker in the left-hand pane of the Execution, Source File, or Disassembly window. If the line in which the breakpoint is set contains several functions, the breakpoint is set on the function that you selected in step 1.

To set or edit a complex breakpoint on a function:

1. Display a list of function names in the Function Names window.
2. Select **Set or Edit Breakpoint** from the Function Names window menu.
3. The Set or Edit Breakpoint dialog is displayed. Complete or alter the details of the breakpoint.
4. Click **OK**.

To set or edit a breakpoint on a low-level symbol:

1. Display the Low Level Symbols window.
2. Select **Set or Edit Breakpoint** from the window menu.
3. Complete or alter the details of the breakpoint.
4. Click **OK**.

Removing a breakpoint

There are five methods of removing a breakpoint:

Method 1

1. Double-click on a line containing a breakpoint (highlighted in red) in the Execution window.
2. Click the **Delete** button in the dialog box that appears.

Method 2

1. Single-click on a line containing a breakpoint (highlighted in red) in the Execution window.
2. Right-click on the line.
3. Select **Toggle breakpoint** from the pop-up menu that is displayed.

Method 3

1. Single-click on a line containing a breakpoint (highlighted in red) in the Execution window.
2. Click the **Toggle breakpoint** button in the toolbar, or press the F9 key.

Method 4

1. Select **Breakpoints** from the View menu to display a list of breakpoints in the Breakpoint window.
2. Select the breakpoint you want to remove.
3. Click the **Toggle breakpoint** button or press the Delete key.

Method 5

1. Select **Delete All Breakpoints** from the Execute menu to delete all breakpoints that are set in the currently selected image. **Delete All Breakpoints** is also available in relevant window menus.

9.1.2 Watchpoints

In its simplest form, a watchpoint halts a program when the value stored in a specified register or memory address changes. The watchpoint halts the program at the next statement or machine instruction after the one that triggered the watchpoint.

There are two types of watchpoints:

- a simple watchpoint that stops when a stored value changes
- a complex watchpoint that:
  - stops when a stored value has changed a specified number of times
  - stops when a stored value changes to a specified value.

——— **Note** ———

If you set a watchpoint on a local variable, you lose the watchpoint as soon as you leave the function that uses the local variable.

———————————

Setting a simple watchpoint

To set a simple watchpoint:

1. Select the variable, area of memory, or register you want to watch.
2. Set the watchpoint in any of the following ways:
   - select **Toggle Watchpoint** from the Execute menu
   - select **Toggle Watchpoint** from the window-specific menu
   - click the **Watchpoint** button.

Select **Watchpoints** from the View menu to see all the watchpoints set in your executable image.

Complex watchpoints

When you set a complex watchpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Watchpoint dialog (see Figure 9-2).

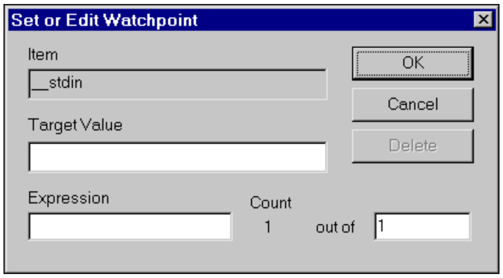


Figure 9-2 Set or Edit Watchpoint dialog

This dialog contains the following fields:

Item The variable or register to be watched (in a read-only field).

Target Value

The value of the variable or register that is to halt the program. If this value is not specified, any change in the value of the item halts the program, dependent on the other watchpoint conditions.

Expression An expression that must be true for the program to halt, in addition to any other watchpoint conditions. Use C-like operators such as:

```
i < 10
i != j
i != j + k
```

Count The program halts when all the watchpoint conditions apply for the *n*th time.

Setting and editing a complex watchpoint

To set a complex watchpoint:

1. Select the variable or register to watch.
2. Select **Set or Edit Watchpoint** from the Execute menu.
3. Specify the required details in the resulting Set or Edit Watchpoint dialog.
4. Click **OK**.

To edit a complex watchpoint:

1. Select **Watchpoints** from the View menu to display current watchpoints.
2. Double-click the watchpoint to edit it.
3. Modify the details as required.
4. Click **OK**.

Removing a watchpoint

Remove a simple watchpoint by using either of the following methods:

Method 1

1. Select **Watchpoints** from the View menu to display a list of watchpoints in the Watchpoint window.
2. Select the watchpoint you want to remove.
3. Remove the selected watchpoint in either of the following ways:
   - click the **Toggle watchpoint** button on the toolbar
   - press the Delete key.

Method 2

1. Position the cursor on a variable or register that has a watchpoint and right-click.
2. Select **Toggle Watchpoint** from the pop-up menu.

Note

If you set a watchpoint on a local variable, you lose the watchpoint as soon as you leave the function that uses the local variable.

9.1.3 Backtrace

When your program has halted, typically at a breakpoint or watchpoint, backtrace information is displayed in the Backtrace window. This displays information about the procedures that are currently active.

The following example shows the backtrace information for a program compiled with debug information and linked with the C library:

```
#DHRY_2:Proc_6 line 42
#DHRY_1:Proc_1 line 315
#DHRY_1:main line 170
PC = 0x0000EB38 (_main + 0x5E0)
PC = 0x0000AE60 (__entry + 0x34)
```

This backtrace provides you with the following information:

Lines 1-3 The first line indicates the function that is currently executing. The second line indicates the source code line from which this function was called, and the third line indicates the call to the second function.

Lines 4-5 Line 4 shows the position of the call to the C library in the main procedure of your program, and the final line shows the entry point in your program made by the call to the C library.

Note

A simple assembly language program assembled without debug information and not linked to a C library shows only the program counter values.

9.1.4 Stepping through an image

To follow the execution of a program more closely than breakpoints or watchpoints allow, you can step through the code. 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.

Step to the next line of code

Step to the next line of code in either of the following ways:

- select **Step** from the Execute menu
- click the **Step** button.

The program moves to the next line of code. This is highlighted in the Execution window. Function calls are treated as one statement. If only C code is displayed, **Step** moves to the next line of C. If disassembled code is shown (possibly interleaved with C source), **Step** moves to the next line of disassembled code.

Step in to a function call

Step in to a function call in either of the following ways:

- select **Step In** from the Execute menu
- click the **Step In** button.

The program moves to the next line of code. If the code is in a called function, the function source appears in the Execution window, with the current line highlighted.

Step out of a function

Step out of a function in either of the following ways:

- select **Step Out** from the Execute menu
- click the **Step Out** button.

The program completes execution of the function and halts at the line immediately following the function call.

Run execution to the cursor

To execute your program to a specific line in the source code:

1. Position the cursor in the line where execution should stop.
2. Select **Run to Cursor** from the Execute menu or click the **Run to Cursor** button.

This executes the code between the current execution and the position of the cursor. Be sure that the execution path includes the statement selected with the cursor.

9.2 ADW and ADU further details

Various debugger windows are described in *ADW and ADU desktop windows* on page 8-11. This section gives more details of some of those windows, and describes other information available to you during a debugging session.

The topics covered in this section are:

- *Working with source files* on page 9-12
- *Working with variables* on page 9-13
- *Displaying disassembled and interleaved code* on page 9-16
- *Remote debug information* on page 9-17
- *Using regular expressions* on page 9-18
- *High-level and low-level symbols* on page 9-19
- *Profiling* on page 9-19
- *Saving an area of memory to disk* on page 9-20
- *Loading an area of memory from disk* on page 9-21
- *Specifying command-line arguments for your program* on page 9-22
- *Using command-line debugger instructions* on page 9-22
- *Changing the data width for reads and writes* on page 9-23
- *Flash download* on page 9-24.

9.2.1 Working with source files

The debuggers provide a number of options that enable you to:

- view the paths that lead to the source files for your program
- list the names of source files that have contributed debug information
- examine the contents of specific source files.

The following sections describe these options in detail.

Search paths

To view the source for your program image during the debugging session, you must specify the location of the files. A search path points to a directory or set of directories that are used to locate files whose location is not referenced absolutely.

If you use the ARM command-line tools to build your project, you might need to edit the search paths for your image manually, depending on the options you chose when you built it.

If you move the source files after building an image, use the Search Paths window to change the search paths set up in ADW or ADU.

To display source file search paths select **Search Paths** from the View menu. The current search paths are displayed in the Search Paths window.

To add a source file search path:

1. Select **Add a Search Path** from the Options menu. The Browse for Folder dialog is displayed.
2. Browse for the directory you want to add and highlight it.
3. Click **OK**.

To delete a source file search path:

1. Select **Search Paths** from the View menu. The Search Paths window is displayed.
2. Select the path to delete.
3. Press the Delete key.

Listing source files

Follow these steps to examine the source files of the current program:

1. Display the Source Files List window, showing the names of the files that have contributed debug information, by selecting **Source Files** from the View menu.
2. Select a source file to examine by double-clicking on its name. The file is opened in its own Source File window.

——— **Note** ———

You can have more than one source file open at a time.

———————————

9.2.2 Working with variables

To display a list of local or global variables, select the appropriate item from the View menu. A Locals/Globals window is displayed. You can also display the value of a single variable, or you can display additional variable information from the Locals/Globals window.

Follow these steps to display the value of a single variable:

1. Select **View** → **Variables** → **Expression**.
2. Enter the name of the variable in the View Expression dialog.
3. Click **OK**. The variable and its value are displayed in the Expression window.

Alternatively:

1. Highlight the name of the variable.
2. Select **View** → **Variables** → **Immediate Evaluation**, or click the **Evaluate Expression** button. The value of the variable is displayed in a message box and in the Command window.

——— **Note** ———

If you select a local variable that is not in the current context, an error message is displayed.

———————————

Changing the value

To change the value of a variable that is displayed in a Local/Globals window, double-click on its current value. In-place editing is invoked whenever possible, otherwise a dialog is displayed allowing you to edit the value.

If the type of the variable is long long or unsigned long long, your new value might be of such a length that it appears to be invalid. In such a case, enter `LL` or `ULL` as appropriate at the end of the new value to force its acceptance.

Changing display formats

If the currently active window is the Locals, Globals, Expressions, or Debugger Internals window, you can change the display format for one or all of the variables.

Follow these steps to change the display format:

1. Right-click on a variable and select **Change line format** (to change the display format for that line only) or **Change window format** (for all lines) from the window menu. The Display Format dialog is displayed.
2. Enter the display format. Use the same syntax as a `printf()` format string in C. Table 9-1 on page 9-15 lists the valid format descriptors.
3. Click **OK**.

Table 9-1 Display formats

Type	Comment	Format	Description
`int`	Use these for integer results only	`%d`	Signed decimal integer (default for integers)
		`%u`	Unsigned integer
		`%x`	Hexadecimal (lowercase letters).
`char`	Use this for a character result only	`%c`	Character.
char*	Use this for expressions that yield a pointer to a null terminated string only	`%s`	Pointer to character.
`void*`	This is safe with any kind of pointer	`%p`	Pointer (`0x%.8lx`), for example, `0x00018ABC`.
`float`	Use these for floating-point results only	`%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`.

Note

Individual line formats are overridden by a change to the window format. A line format of null returns the format of that line to the current window format. A window format of null returns all display formats to the default setting.

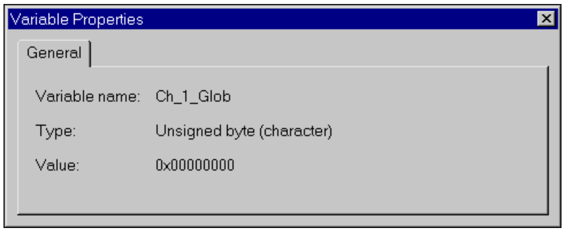
The initial display format of a variable declared as `char[]=` is special. The whole string is displayed, whereas normally arrays are displayed as ellipses (…). If the format is changed it reverts to the standard array representation.

Alternative methods of changing the default display formats for all windows are:

- select **Change Default Display Formats...** from the Options menu and change any of the displayed format strings
- select **Debugger Internals** from the View menu and change the value of variables such as `uint_format`, `float_format`, and so on.

Variable properties

If you have a list of variables displayed in a Locals/Globals window, you can display additional information on a variable by selecting **Properties** from the window-specific menu (see Figure 9-3). To display the window-specific menu, right-click on an item. The information is displayed in a dialog.



Variable name: Ch_1_Glob
Type: Unsigned byte (character)
Value: 0x00000000

Figure 9-3 Variable Properties dialog

Indirection

Select **Indirect through item** from the Variables menu to display other areas of memory.

If you select a variable of `integer` type, the value is converted to a pointer. Sign extension is used if applicable, and the memory at that location is displayed. If you select a pointer variable, the memory at the location pointed to is displayed. You cannot select a `void` pointer for indirection.

9.2.3 Displaying disassembled and interleaved code

You can display disassembled code in the Execution window or in the Disassembly window. Select **Disassembly** from the View menu to display the Disassembly window.

You can choose the type of disassembled code to display by selecting the **Disassembly mode** submenu from the Options menu. ARM code, Thumb code, or both can be displayed, depending on your image.

To display interleaved C or C++ and assembly language code:

1. Select **Toggle Interleaving** from the Options menu to display interleaved source and assembly language in the Execution window. Disassembled code is displayed in grey. The C or C++ code is displayed in black.

To display an area of memory as disassembled code:

1. Select **Disassembly** from the View menu, or click the **Display Disassembly** button. The Disassembly Address dialog is displayed.
2. Enter an address or an expression, for example `@main`.
3. Click **OK**. The Disassembly window displays the assembler instructions derived from the code held in the specified area of memory. Use the scroll bars to display the content of another memory area, or:
   a. Select **Goto** from the Search menu.
   b. Enter an address.
   c. Click **OK**.

Specifying a disassembly mode

ADW and ADU try to display disassembled code as ARM code or Thumb code, according to settings encoded in the debug information. Sometimes, however, the type of code required cannot be determined. This can happen, for example, if you have copied the contents of a disk file into memory or if you are disassembling a ROM.

When you display disassembled code in the Execution window you can choose to display ARM code, Thumb code, or both. To specify the type of code displayed, select **Disassembly mode** from the Options menu.

9.2.4 Remote debug information

The RDI Log window displays the low-level communication messages between the debugger and the target processor.

This facility is not normally enabled. It must be specially turned on when the RDI is compiled.

To display remote debug information, select **RDI Protocol Log** from the View menu. The RDI Log window is displayed.

Use the RDI Log Level dialog, obtained by selecting **Set RDI Log Level** from the Options menu, to select the information to be shown in the RDI Log window:

Bit 0	RDI level logging on or off.
Bit 1	Device driver logging on or off.

9.2.5 Using regular expressions

Use regular expressions to specify and match strings. A regular expression is either:

- a single extended ASCII character (other than the special characters described below)
- a regular expression modified by one of the special characters.

You can include low-level symbols or high-level symbols in a regular expression (see *High-level and low-level symbols* on page 9-19 for more information).

Pattern matching follows the UNIX `regexp(5)` format, but without the special symbols, `^` and `$`.

The following special characters modify the meaning of the previous regular expression (and work only when they follow a regular expression):

`*`	Zero or more of the preceding regular expressions. For example, `A*B` matches `B`, `AB`, and `AAB`.
`?`	Zero or one of the preceding regular expression. For example, `AC?B` matches `AB` and `ACB` but not `ACCB`.
`+`	One or more of the preceding regular expression. For example, `AC+B` matches `ACB` and `ACCB`, but not `AB`.

The following special characters are regular expressions in themselves:

`\`	Precedes any special character you need to include literally in an expression to form a single regular expression. For example, `\*` matches a single asterisk (`*`) and `\\` matches a single backslash (`\`). The regular expression `\x` is equivalent to `\x` as the character `x` is not a special character.
`()`	Allows grouping of characters. For example, `(202)*` matches `202202202` (as well as nothing at all), and `(AC?B)+` looks for sequences of `AB` or `ACB`, such as `ABACBAB`.
`.`	Exactly one character. This is different from `?` in that the period (`.`) is a regular expression in itself, so `.*` matches all, while `?*` is invalid. The special character `.` does *not* match the end-of-line character.
`[ ]`	A set of characters, any one of which can appear in the search match. For example, the expression `r[23]` matches strings `r2` and `r3`. The expression `[a-z]` matches all characters between `a` and `z`.

9.2.6 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 the function name shown in the Function Names window.

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.

You can display a list of the low-level symbols in your program in the Low Level Symbols window.

In a regular expression, precede the symbol with `@` to indicate a low-level symbol.

9.2.7 Profiling

Profiling involves sampling the *program counter* (pc) at specific time intervals. From this information you can estimate the percentage of time spent in each procedure. Using the `armprof` command-line program on the data generated by ADW or ADU, you see where effort can be most effectively spent to make the program more efficient.

——— **Note** ———

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

———

To collect profiling information:

1. Load your image file.
2. Select **Options** → **Profiling** → **Toggle Profiling**.
3. Execute your program.
4. When the image terminates, select **Options** → **Profiling** → **Write to File**.
5. A Save dialog appears. Enter a file name and a directory as necessary.
6. Click **Save**.

——— **Note** ———

You cannot display profiling information from within the debugger. You must capture the data using the **Profiling** functions on the Options menu, then use the `armprof` command-line tool, described in the *ADS Compiler, Linker, and Utilities Guide*.

———

Profiling information is collected from the beginning of program execution. If you want to collect information on just a part of the execution:

1. Initiate collection of profiling information before executing the program.
2. Clear the information collected up to a breakpoint at the beginning of the region of interest, by selecting **Options** → **Profiling** → **Clear Collected**.
3. Execute the program as far as another breakpoint at the end of the region of interest.

9.2.8 Saving an area of memory to disk

To copy the contents of an area of memory to a disk file:

1. Select **Put File** from the File menu to display the Put file dialog (see Figure 9-4).

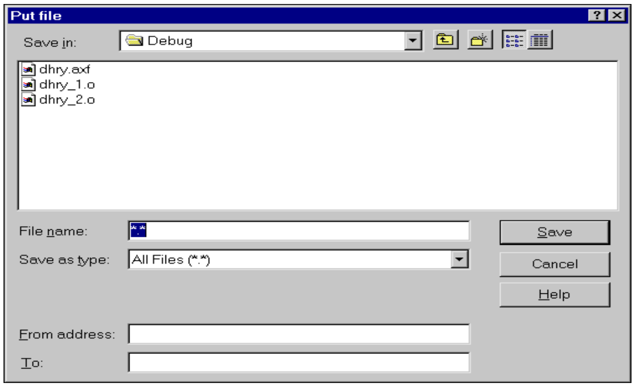


Figure 9-4 Put File dialog

2. Enter the name of the file to write to.
3. Enter a memory area in the From address and To fields.
4. Click **Save**.
5. Click **OK**. The output is saved as a binary file.

9.2.9 Loading an area of memory from disk

To copy the contents of a disk file to memory:

1. Select **Get File** from the File menu to display the Get file dialog (Figure 9-5).

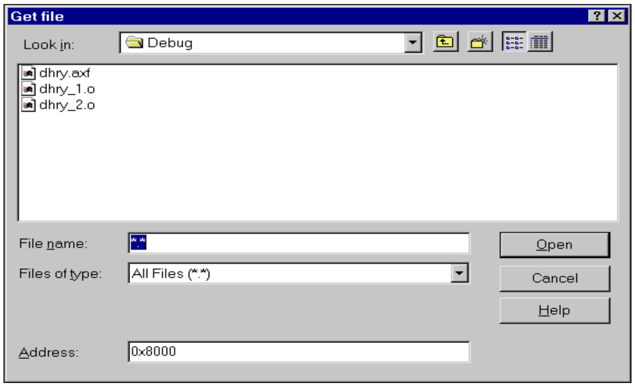


Figure 9-5 Get File dialog

2. Select the file you want to load into memory.
3. Enter a memory address where the file must be loaded.
4. Click **Open**.

9.2.10 Specifying command-line arguments for your program

Follow these steps to specify the command-line arguments for your program:

1. Select **Set Command Line Args** from the Options menu. The Command Line Arguments dialog is displayed (see Figure 9-6).

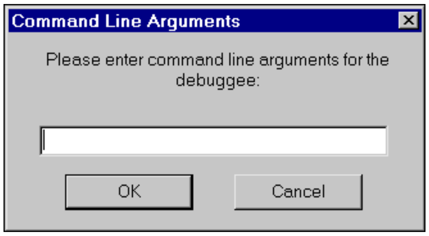


Figure 9-6 Command Line Arguments dialog

2. Enter the command-line arguments for your program.

3. Click **OK**.

Note

You can also specify command-line arguments when you load your program in the Open File dialog or by changing the debugger internal variable, `$cmdline`.

9.2.11 Using command-line debugger instructions

If you are familiar with the *ARM Symbolic Debugger* (armsd) you might prefer to use almost the same set of commands from the Command window. The armsd command `Pause` is unavailable in the Command window. Follow these steps to use all other armsd commands from within ADW or ADU:

1. Select **Command** from the View menu to open the Command window displaying a `Debug:` command line.

2. Enter ARM command-line debug commands at this prompt. The syntax used is the same as for armsd. Type `help` for information on the available commands.

Refer to Part C of this book for more information on armsd.

9.2.12 Changing the data width for reads and writes

You can use the Command window to enter a command that reads data from, or writes data to memory. You must, however, be aware of the default width of data read or written, and how to change it if necessary. By default, a read from or write to memory in ADW or ADU transfers a *word* value. For example:

```
let 0x8000 = 0x01
```

transfers 4 bytes to memory starting at address `0x8000`. In this example the bytes at `0x8001`, `0x8002`, and `0x8003` are all zero-filled.

To write a single byte to memory, use an instruction of the form:

```
let *(char *) 0xaddress = value
```

To read a single byte from memory, use an instruction of the form:

```
print /%x *(char *) 0xaddress
```

where `/%x` means *display in hexadecimal*.

You can also read and write halfword **`short`** values in a similar way, for example:

```
let *(short *) 0xaddress = value
print /%x *(short *) 0xaddress
```

You can also select **View** → **Variables** → **Expression** to open the View Expression window, and use that to specify bytes or shorts for displaying memory. For example:

- for bytes, enter `*(char *) 0xaddress` in the View Expression box
- for halfwords, enter `*(short *) 0xaddress` in the View Expression box.

To display in hexadecimal, click the right mouse button on the Expression window, select **Change Window Format** and enter `%x`.

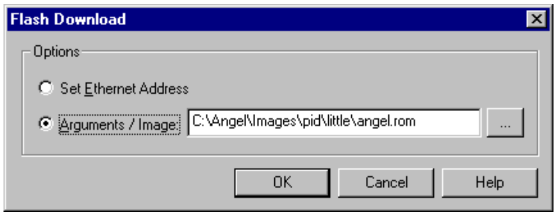
——— **Note** ———

Changes to window formats are saved. Changes to line formats are not saved. If you select **Change Window Format** and leave the format field blank, the setting defaults to the original setting.

———————————

9.2.13 Flash download

Use the Flash Download dialog (see Figure 9-7) to write an image to the Flash memory chip on an ARM Development Board or any suitably equipped hardware.



Arguments / Image: C:\Angel\Images\pid\little\angel.rom

Figure 9-7 Flash Download dialog

Set Ethernet Address

Use the **Set Ethernet Address** option if necessary after writing an image to Flash memory. You might do this, for example, if you are using Angel with Ethernet support.

When you click **OK**, you are prompted for the IP address and netmask, for example, 193.145.156.78.

You do not need to use this option if you have built your own Angel port with a fixed Ethernet address.

Arguments / Image

...

Specifies the arguments or image to write to Flash. Use the **Browse** button to select the image.

For more information about writing to Flash memory, including details of how to build your own Flash image, refer to the *ADS Debug Target Guide* and the *ADS Compiler, Linker, and Utilities Guide*.

9.3 Communications channel viewers

ADW supports the use of viewers to access debug communication channels. An example communications channel viewer is supplied with ADW (`ThumbCV.dll`) or you can provide your own viewer.

Note

ADU also supports the use of comms channel viewers, but ADS does not yet include a comms channel viewer that runs under UNIX.

9.3.1 ThumbCV communications channel viewer

To select a communications channel viewer when running ADW:

1. Select **Configure Debugger** from the Options menu.
2. On the Target tab, select **Remote_A**.
3. Click the **Configure** button. The Remote_A Connection dialog is displayed.
4. Select **Channel Viewer Enabled**. The **Add** and **Remove** buttons are activated.
5. Click the **Add** button, navigate if necessary to the `bin` subdirectory of the ADS installation directory, and a list of `.DLLs` is displayed (provided that **Windows Explorer** → **View** → **Options...** → **Show all files** is selected).
6. Select the appropriate `.DLL` and click the **Open** button.

   Click the **OK** button on either the Remote_A Connection dialog or the Debugger Configuration dialog to restart ADW with an active comms channel viewer. See *Remote_A connection* on page 9-38 for more information on the Remote_A Connection dialog. `ThumbCV.DLL` provides the viewer shown in Figure 9-8.

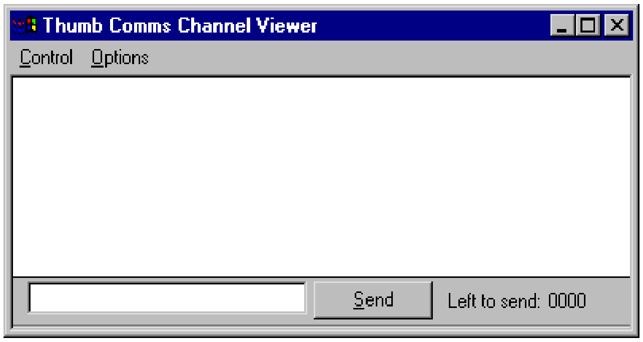


Figure 9-8 Thumb Comms Channel Viewer

This window has a dockable dialog bar at the bottom that is used to send information down the channel. Typing information in the edit box and clicking the **Send** button stores the information in a buffer. The information is sent when requested by the target. The Left to send counter displays the number of bytes that are left in the buffer.

Sending information

To send information to the target, type a string into the edit box on the dialog bar and click the **Send** button. The information is sent when requested by the target, in ASCII character codes.

Receiving information

The information that is received by the channel viewer is converted into ASCII character codes and displayed in the window, if the channel viewers are active. However, if `0xFFFFFFFF` is received, the following word is treated and displayed as a number.

9.4 Debugger configuration

This section describes the three tabbed screens of the Debugger Configuration dialog:

- *Target environment* on page 9-27
- *Debugger* on page 9-29
- *Memory Maps* on page 9-30.

Note

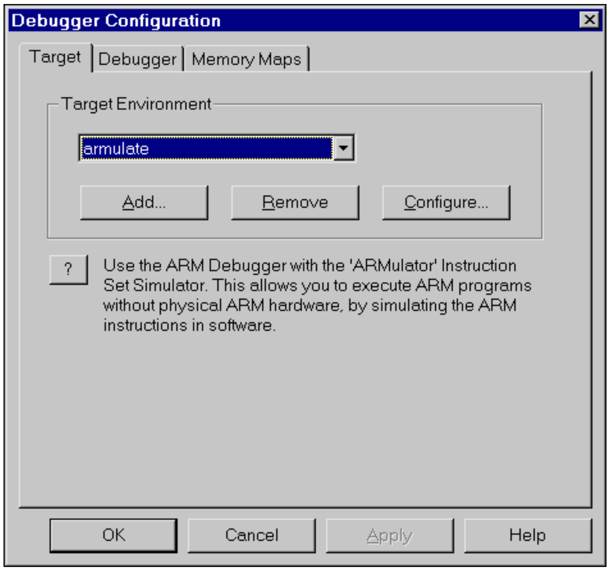
In some of these procedures you need to locate and select a required file. A browse dialog helps you do this. However, files of the type you need are not listed unless **Windows Explorer** → **View** → **Options...** → **Show all files** is selected.

Select **Configure Debugger** from the Options menu to open the Debugger Configuration dialog.

9.4.1 Target environment

To configure the target environment:

1. Click the **Target** tab of the Debugger Configuration dialog (see Figure 9-9).



Use the ARM Debugger with the 'ARMulator' Instruction Set Simulator. This allows you to execute ARM programs without physical ARM hardware, by simulating the ARM instructions in software.

Figure 9-9 Configuration of target environment

2. Change the following configuration options, as required:

   Target Environment
   Select the target environment for the image being debugged.

   Add Display an Open dialog to add a new environment to the debugger configuration.

   Remove Remove a target environment.

   Configure
   Display a configuration dialog for the selected environment (as described in *Target Configuration* on page 9-33).

   ? Display a more detailed description of the selected environment.

3. Save or discard your changes:
   - click **OK** to save any changes and exit
   - click **Apply** to save any changes
   - click **Cancel** to ignore all changes not applied and exit
   - click **Help** to display online help.

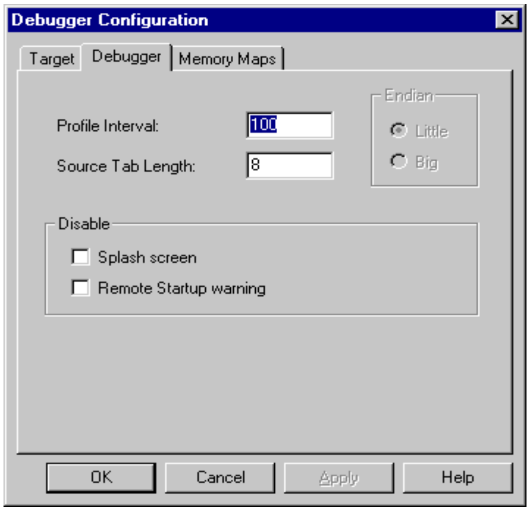
——— **Note** ———

Apply is disabled for the Target page because a successful RDI connection must be made first. When you click **OK** an attempt is made to make your selected RDI connection. If this does not succeed, the ARMulate setting is restored.

9.4.2 Debugger

To change the configuration used by the debugger:

1. Click the **Debugger** tab of the Debugger Configuration dialog (see Figure 9-10).



Profile Interval: 100
Source Tab Length: 8
Endian: [✓ Little] Big
Disable:
☐ Splash screen
☐ Remote Startup warning

Figure 9-10 Configuration of debugger

2. Change the following configuration settings, as required:

Profile Interval

This is the time between pc sampling in microseconds. It is applicable to ARMulator and Angel only. Lower values give more accurate results than higher values, but slow down execution more.

Source Tab Length

This specifies the number of space characters used for tabs when displaying source files.

Endian Use these buttons to inform the debugger that the target is operating in little-endian or big-endian mode.

Little	Low addresses have the least significant bytes.
Big	High addresses have the least significant bytes.

These buttons are disabled if you are using RDI 1.51. In that case similar buttons are enabled on the target configuration dialog.

Disable Allows you to turn off the following display features:

Splash screen

When selected, stops display of the splash screen (the ARM Debugger startup box) when the debugger is first loaded.

Remote Startup warning

Turns on or off the warning that debugging is starting with a session requiring debug hardware.

If the warning is turned off and debugging is started without the necessary hardware attached, there is a possibility that ADW or ADU might hang.

If the warning is enabled, you have the opportunity to start with ARMulator.

3. Save or discard your changes:
   - click **OK** to save any changes and exit
   - click **Apply** to save any changes
   - click **Cancel** to ignore all changes not applied and exit.

Note

When you make changes to the debugger configuration the current execution is ended and your program is reloaded.

9.4.3 Memory Maps

For the version of ARMulator that is supplied as part of ADS you can select a memory map file in the ARMulator Configuration dialog (see *ARMulator configuration* on page 9-33).

For older versions of ARMulator only (for example, the version supplied with SDT 2.50), the matching older configuration file `armul.cnf` must also be present in the `\bin` directory, and you can configure Memory Maps as follows:

1. Click the **Memory Maps** tab of the Debugger Configuration dialog (see Figure 9-11 on page 9-31).

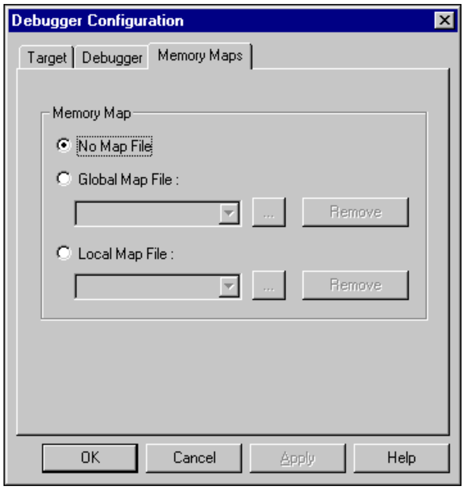


Figure 9-11 Configuration of ARM Debugger memory maps

2. Change the following configuration settings, as required:

   Memory Map

   This allows you to specify a memory map file, containing information about a simulated memory map that ARMulator uses. It applies to older versions of ARMulator only. The file includes details of the databus widths and access times for each memory region in the simulated system. See the *ADS Debug Target Guide* for more information.

   You can select one of three Memory Map options:

   No Map File

   Use the ARMulator default memory map. This is a flat 4GB bank of ideal 32-bit memory, having no wait states.

   Global Map File

   Use a global memory map. Select this option to use the specified memory map file for every image loaded during the current debug session.

   A box allows you to enter a filename or to select a filename from a pull-down list. Use this box to add new map files to the list, or select a map file from the list. When you have selected a map file, the debugger checks that the file exists and is of a valid format. Any file that fails these checks is removed from the list. The dialog remains, however, so you can correct an error or select another map file if necessary.

   Use the **Remove** button to remove the currently selected file from the list.

The browse button allows you to select a memory map file using a dialog.

Local Map File

Use a local memory map. Select this option to use a memory map file that is local to a project.

If a local memory map file is required when the debugger is initialized, the current working directory is searched. If a re-initialization occurs after the debugger has started and loaded an image, the directory containing the image is searched.

A box allows you to pull down a list of filenames, add a new filename to the list, or select a filename from the list. You must not specify an *absolute* path name, but you can specify a memory map file *relative* to the current image path.

The browse button allows you to select a memory map file using a dialog.

When you have specified a filename, the debugger does not check for the existence of the file or the validity of its format. If the format of the file is found to be invalid at re-initialization, the debugger displays an error message. In that case, or if the file does not exist, the debugger defaults to the No Map File option and uses the ARMulator default settings.

Use the **Remove** button to remove the currently selected file from the list.

Note

Map files are used only at re-initialization, not when a program is loaded. When you select the Local Map File option, the map file in the working directory of the current image is used. If you load a new image, the same map file is used. To use a map file that is associated with the new image, you must re-initialize the debugger by selecting **Configure Debugger…** from the Options menu and clicking **OK**.

3. Save or discard your changes:
   - click **OK** to save any changes and exit
   - click **Apply** to save any changes
   - click **Cancel** to ignore all changes not applied and exit.

9.5 Target Configuration

The target configurations you can perform are described under:

- *ARMulator configuration* on page 9-33
- *Remote_A connection* on page 9-38
- *Multi-ICE configuration* on page 9-40
- *EmbeddedICE configuration* on page 9-41
- *Gateway configuration* on page 9-42.

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 need are not listed unless **Windows Explorer** → **View** → **Options...** → **Show all files** is selected.

9.5.1 ARMulator configuration

To change configuration settings for ARMulator:

1. Select **Configure Debugger** from the Options menu.
2. Click on the **Target** tab.
3. Select **ARMulate** in the Target Environment field.
4. Click on the **Configure** button. Two ARMulator Configuration dialogs are available. The one appropriate for the ARMulator you are using is displayed.

   Descriptions follow of:

   - *Configuration of newer ARMulator* on page 9-34
   - *Configuration of older ARMulator* on page 9-36.

   Be sure to read only the one that applies to you.
5. When you are satisfied with all the settings, click **OK**.

Configuration of newer ARMulator

You normally use the ARMulator supplied in file `armulate.dll` as part of the ARM Developer Suite. The ARMulator Configuration dialog appears as shown in Figure 9-12.

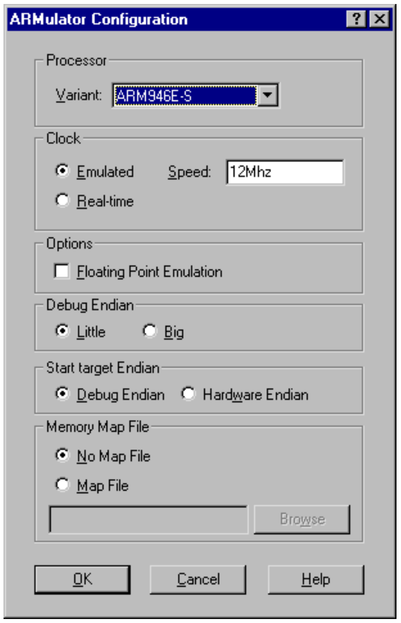


Figure 9-12 Configuration of newer ARMulator

The configuration dialog of the newer ARMulator allows you to examine and change the following settings:

Processor	Specify which ARM processor you want ARMulator to simulate.
Clock	Choose between simulating a processor clock running at a speed that you can specify, or executing instructions in real time.
Options	Specify whether floating point arithmetic is to be emulated.

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.

Start target Endian

Select the way in which the byte order of ARMulator models that have a CP15 coprocessor is determined:

- Select Debug Endian to instruct the model to use the byte order set by the Debug Endian button.
- Select Hardware Endian 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.

Memory Map File

Specify a memory map file, or that you want to use default settings.

For information about ARMulator Clock speed settings, refer to *ARMulator clock speed* on page 9-37.

If you are using the software floating-point C libraries, ensure that the Floating Point Emulation option is **off** (blank). The option should be **on** (checked) only if you are using the *Floating Point Emulator* (FPE).

If, in the Memory Map File box, 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 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*.

Configuration of older ARMulator

If you are using an ARMulator (`armulate.dll` file) older than the one supplied as part of ADS, the ARMulator Configuration dialog appears as shown in Figure 9-13.

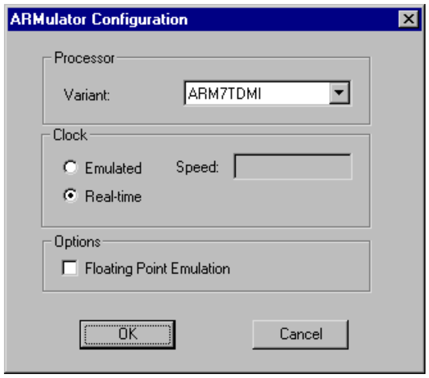


Figure 9-13 Configuration of older ARMulator

The configuration dialog for the older ARMulator enables you to:

- specify which ARM processor you want ARMulator to simulate
- choose between simulating a processor clock running at a speed that you can specify, or executing instructions in real time
- specify whether floating-point arithmetic is to be emulated.

For information about ARMulator Clock speed settings, refer to *ARMulator clock speed* on page 9-37.

If you are using the software floating-point C libraries, ensure that the Floating Point Emulation option is **off** (blank). The option should be **on** (checked) only if you are using the FPE.

Note

If you use the SDT `armulate.dll` file, you must use the SDT `armul.cnf` file. If you use the ADS `armulate.dll` file, you must use the ADS `default.ami` file.

ARMulator clock speed

If you set a nonzero simulated Clock Speed, then the clock speed used is the value that you enter. Values stored in debugger internal variable `$clock` depend on this setting, and are unavailable if you set the speed to `0.00` (older ARMulator) or select **Real-time** (newer ARMulator). For information about debugger internal variables, see *Debugger Internals window* on page 8-15. The ADW or ADU clock speed defaults to 0.00 for compatibility with the defaults of armsd. Setting 0.00MHz or selecting **Real-time** in ADW or ADU 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.

For ARMulator, an unspecified clock frequency is of no consequence because ARMulator does not require a clock frequency to be able to simulate the execution of instructions and count cycles (for `$statistics`). However, your application program might sometimes need to access a clock, so ARMulator must always be able to give clock information. That is why the clock frequency from the configuration file is used by ARMulator if no simulated clock speed is specified.

In either case, the clock information is used by ARMulator 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 `SWI_clock`, and is also visible to the user from the debugger as `$clock`. It is also used internally by ADW, ADU, and armsd in the calculation of `$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.

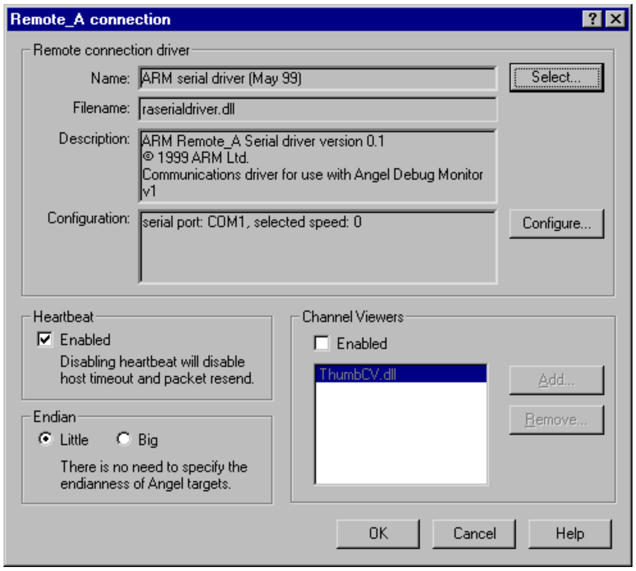
`$memstats` is handled slightly differently because it does require a defined clock frequency, so that ARMulator can calculate how many wait states are needed for the memory speed defined in an `armsd.map` file. If a clock speed is specified and an `armsd.map` file is present, then `$memstats` can give useful information about memory accesses and times. Otherwise, for the purposes of 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.

9.5.2 Remote_A connection

If you are using Angel or EmbeddedICE, use the Remote_A connection dialog to configure the settings for the remote connection you are using to debug your application.

To change remote connection settings:

1. Select **Configure Debugger** from the Options menu.
2. Click on the Target tab.
3. Select **Remote_A** Target Environment to select *Angel Debug Protocol* (ADP).
4. Click **Configure** to display the Remote_A connection dialog (see Figure 9-14).
5. When you are satisfied with any changes you make to the settings, click **OK**.


Remote_A connection

Remote connection driver

Name: ARM serial driver (May 99)

Filename: raserialdriver.dll

Description: ARM Remote_A Serial driver version 0.1
© 1999 ARM Ltd.
Communications driver for use with Angel Debug Monitor v1

Configuration: serial port: COM1, selected speed: 0

Heartbeat
[✓] Enabled
Disabling heartbeat will disable host timeout and packet resend.

Endian
(● Little) ○ Big
There is no need to specify the endianness of Angel targets.

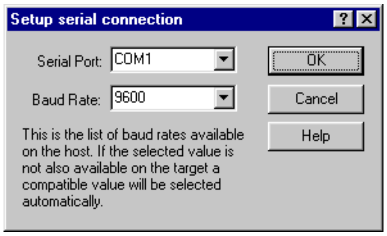
Channel Viewers
[ ] Enabled
ThumbCV.dll

Figure 9-14 Configuration of Remote_A connection

The Remote_A connection dialog allows you to examine and change:

Remote connection driver

Click **Select...** to see a list of available drivers, including Serial, Serial /Parallel, and Ethernet. Select one to use it instead of the current driver. To change the settings of the currently selected driver, click **Configure...**. A dialog appears, similar to one of Figure 9-15, Figure 9-16, or Figure 9-17.

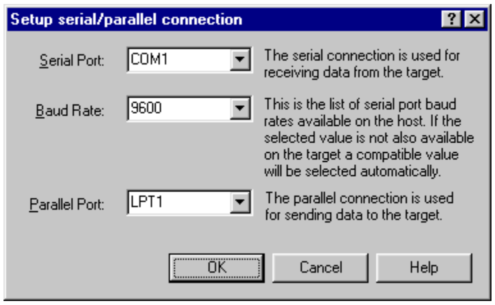

Setup serial connection

Serial Port: COM1

Baud Rate: 9600

This is the list of baud rates available on the host. If the selected value is not also available on the target a compatible value will be selected automatically.

Figure 9-15 Serial connection configuration

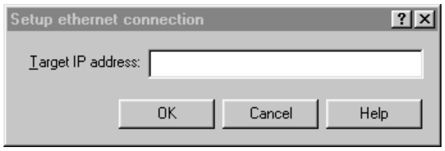

Setup serial/parallel connection

Serial Port: COM1 — The serial connection is used for receiving data from the target.

Baud Rate: 9600 — This is the list of serial port baud rates available on the host. If the selected value is not also available on the target a compatible value will be selected automatically.

Parallel Port: LPT1 — The parallel connection is used for sending data to the target.

Figure 9-16 Serial/parallel connection configuration


Setup ethernet connection

Target IP address:

Figure 9-17 Ethernet connection configuration

Heartbeat Ensures reliable transmission by sending heartbeat messages. Any errors are more easily detected when known messages are expected regularly.

Endian Use these buttons to inform the debugger that the target is operating in little-endian or big-endian mode. Generally, Angel can make the correct endian setting in this dialog automatically.

These buttons are disabled if you are using RDI 1.50. In that case similar buttons are enabled on the Debugger tabbed page of the Debugger Configuration dialog.

Channel Viewers

Channel viewers are not supported by ADU.

In ADW, checking Enabled allows you to add, remove, or select channel viewers in the displayed list of `.dll` files. ARM supplies a channel viewer for use with ADW in file `ThumbCV.dll`. See *ThumbCV communications channel viewer* on page 9-25 for more information.

Click the **Add...** button to add a channel viewer DLL to the displayed list.

Click the **Remove...** button to remove the currently selected channel viewer DLL from the displayed list.

9.5.3 Multi-ICE configuration

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)
- the selection of a channel viewer.

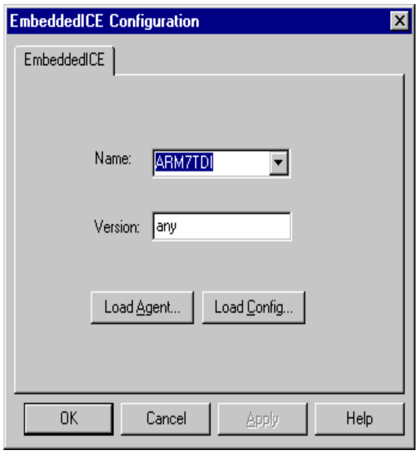
Full descriptions of Multi-ICE configuration are given in the Multi-ICE documentation and in the online help available when the dialog is displayed.

9.5.4 EmbeddedICE configuration

Use the EmbeddedICE Configuration dialog to select the settings for an EmbeddedICE target. This option is enabled only if EmbeddedICE is connected to your machine.

To change the EmbeddedICE configuration options:

1. Select **Configure EmbeddedICE** from the Options menu. A configuration dialog, shown in Figure 9-18, is displayed.



Name: ARM7TDI
Version: any

Figure 9-18 Configuration of EmbeddedICE target

2. Change the following configuration settings, as required:

Name Name given to the EmbeddedICE configuration. Valid options are:

ARM7DI For use with ARM7 core with debug extensions and EmbeddedICE logic (includes ARM7DMI).

ARM7TDI

For use with ARM7 core with Thumb and debug extensions and EmbeddedICE logic (includes ARM7TDMI™).

Version Version given to the EmbeddedICE configuration. Specify the version to use or enter `any` if you do not require a specific implementation.

Load Agent

Specify a new EmbeddedICE ROM image file, download it to your board, and run it. Use this for minor updates to the ROM.

Load Config

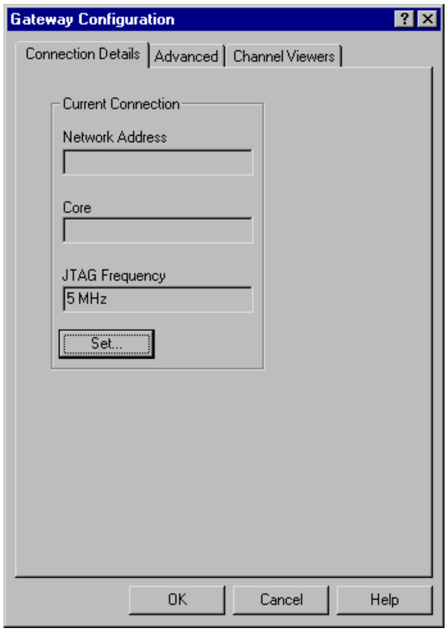
Specify an EmbeddedICE configuration file to load. Click **OK** to run.

9.5.5 Gateway configuration

If you need to add Gateway to the list of available targets, click **Add** and use the resulting browse dialog to locate and select the `gateway.dll` file.

Select the Gateway target line and click the **Configure** button to display the dialog shown in Figure 9-19. The Gateway Configuration dialog has the following three tabbed pages:

- Connection Details
- Advanced
- Channel Viewers.



5 MHz

Figure 9-19 ARM Gateway Configuration dialog

On the Connection Details tabbed page, click **Set** to set connection details for your probe. The resulting Set Connection Details dialog is shown in Figure 9-20 on page 9-43.

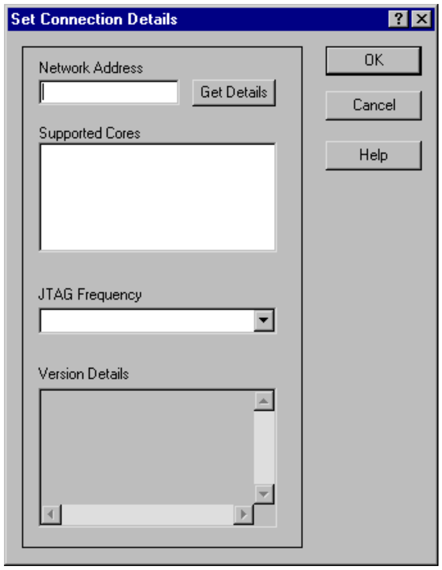


Figure 9-20 Gateway Configuration, Set Connection Details dialog

1. Enter the IP address for your probe in the IP Address field. See your HP documentation for more information on setting the IP address.

2. Click on the Lookup button. The Debugger establishes a connection with probe and displays a list of supported targets in the Supported Cores field.

3. Select the target type you want in the Supported Cores field.

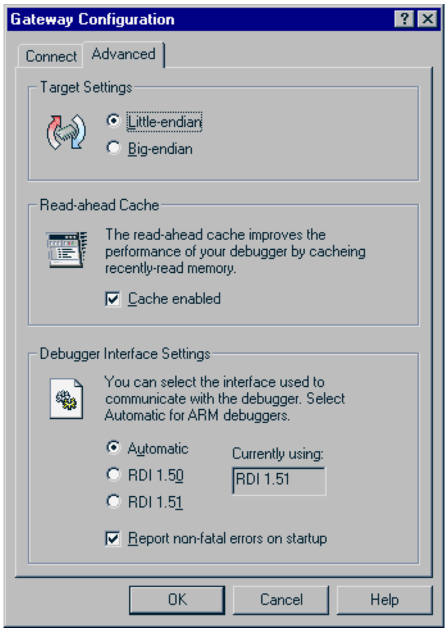
4. Select the JTAG base clock speed you require from the JTAG Frequency drop-down menu. This sets the frequency at which the probe clocks data across the target JTAG port. Higher frequencies give improved performance, especially for JTAG-intensive operations such as downloading.

   You are recommended to select the highest frequency supported by your target hardware. Hardware constraints such as stacking several devices together, or using long cables between the probe and the target, might require you to use lower JTAG frequencies.

5. Click **OK** to confirm your settings and close the Set Connection Details panel.

The Advanced tabbed page of the Gateway Configuration dialog, shown in Figure 9-21, contains three areas:

- Target Settings
- Read-ahead Cache
- Debugger Interface Settings.


Gateway Configuration

Connect | Advanced

Target Settings

[✓ Little-endian] Big-endian

Read-ahead Cache

The read-ahead cache improves the performance of your debugger by cacheing recently-read memory.

[✓ Cache enabled]

Debugger Interface Settings

You can select the interface used to communicate with the debugger. Select Automatic for ARM debuggers.

[✓ Automatic] RDI 1.50 RDI 1.51

Currently using: RDI 1.51

[✓ Report non-fatal errors on startup]

Figure 9-21 Gateway Configuration, Advanced tab

Normally these settings are correct, but you can change them if required.

The Target Settings area shows the byte order of the specified target:

Little-endian Denotes a target that has the least significant byte of each word at the lowest memory address.

Big-endian Denotes a target that has the most significant byte of each word at the lowest memory address.

The Read-ahead Cache option is enabled by default. Read-ahead cacheing improves memory read performance by reading more memory than requested by the debugger. The additional memory is cached in case it is needed later.

If you are debugging a system with demand-paged memory, deselect the Cache enabled option.

The radio buttons in the Debugger Interface Settings area allow you to specify the version of Remote Debugging Interface (RDI) you want Gateway to use for communicating with the debugger. Automatic is the default setting. Your debugger and Gateway determine the correct standard automatically. You can use this setting for all the ARM debuggers. If you need to specify a particular version of RDI, select either RDI 1.50 or RDI 1.51. The RDI version in use is displayed in Currently using.

By default, Gateway reports errors as you connect to the target. If non-fatal error reports prevent you from starting a debugging session, you can stop them being reported by deselecting the Report non-fatal errors on startup check box.

The Channel Viewers tabbed page of the Gateway Configuration dialog is shown in Figure 9-22.

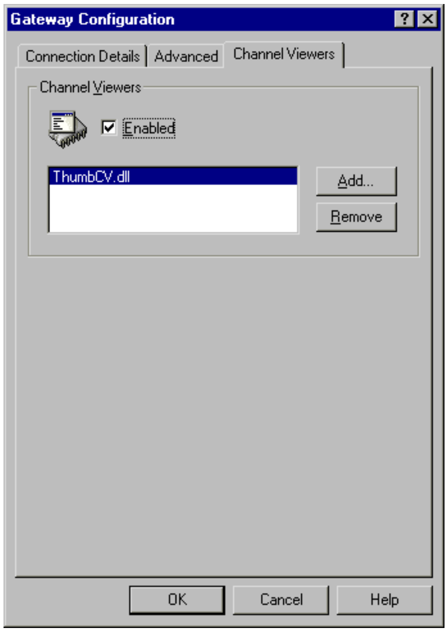


Figure 9-22 Gateway Configuration, Channel Viewers tab

If the selected core supports the Debug Communications Channel (DCC), you can select a channel viewer.

Select the Enabled checkbox to activate a channel viewer. The display shows you the available channel viewers. To browse and select another channel viewer to add to the list, click the **Add** button. To remove a channel viewer from the list, select it and click the **Remove** button.

Select the channel viewer you want to use from the displayed list.

Chapter 10
Using ADW and ADU with C++

This chapter describes the additions that ARM C++ makes to ADW and ADU. It contains the following sections:

10.1 About ADW and ADU for C++

ADW and ADU support C++ debugging with:

- a C++ menu between the View and Execute menus in the main menu bar
- five buttons in the ADW or ADU toolbar:

  Evaluate Expression

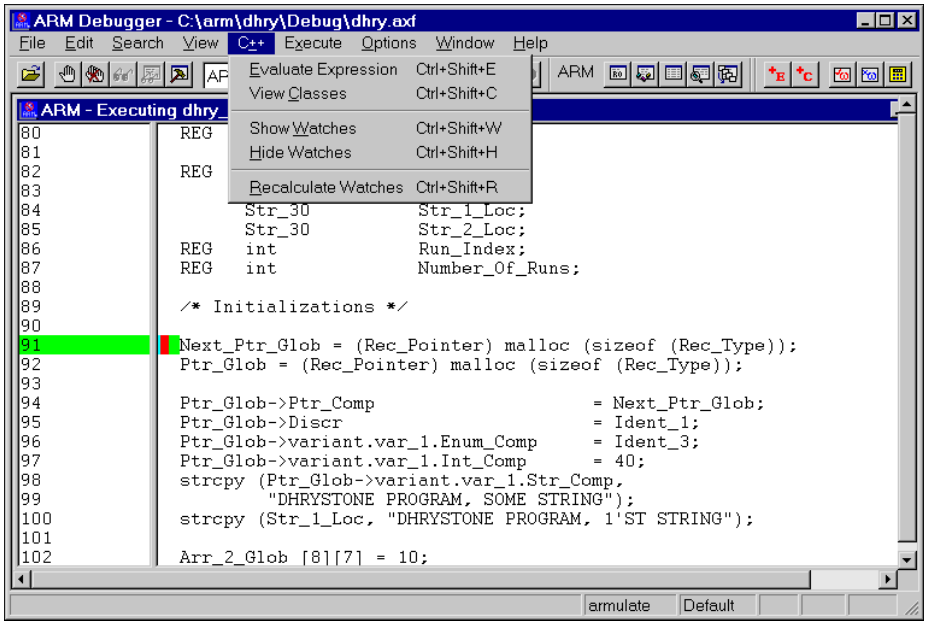
  View Classes

  Show Watches

  Hide Watches

  Recalculate Watches.

Figure 10-1 shows an example of the ADW and ADU C++ debug interface and the C++ menu.


ARM Debugger - C:\arm\dhry\Debug\dhry.axf
File
Edit
Search
View
C++
Execute
Options
Window
Help
Evaluate Expression Ctrl+Shift+E
View Classes Ctrl+Shift+C
Show Watches Ctrl+Shift+W
Hide Watches Ctrl+Shift+H
Recalculate Watches Ctrl+Shift+R
ARM
ARM - Executing dhry_
armulate
Default

```
REG

REG

      Str_30          Str_1_Loc;
      Str_30          Str_2_Loc;
REG   int             Run_Index;
REG   int             Number_Of_Runs;

/* Initializations */

Next_Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));
Ptr_Glob = (Rec_Pointer) malloc (sizeof (Rec_Type));

Ptr_Glob->Ptr_Comp                    = Next_Ptr_Glob;
Ptr_Glob->Discr                       = Ident_1;
Ptr_Glob->variant.var_1.Enum_Comp     = Ident_3;
Ptr_Glob->variant.var_1.Int_Comp      = 40;
strcpy (Ptr_Glob->variant.var_1.Str_Comp,
        "DHRYSTONE PROGRAM, SOME STRING");
strcpy (Str_1_Loc, "DHRYSTONE PROGRAM, 1'ST STRING");

Arr_2_Glob [8][7] = 10;
```

Figure 10-1 The ADW and ADU C++ interface

10.2 Using the C++ debugging tools

The menu items in the C++ menu provide three additional debugger windows:

- *Class View window* on page 10-3 displays the class hierarchy of a C++ program in outline format
- *Watch window* on page 10-5 displays a list of watches, allowing you to add and remove variables and expressions to be watched, and change the contents of watched variables
- *Evaluate Expression window* on page 10-10 allows you to enter an expression to be evaluated, and to add that expression to the Watch window.

10.2.1 Using the Class View window

You can use the Class View window to view the class structure of your C++ program. Classes are displayed in an outline format that allows you to navigate through the hierarchy to display the member functions for each class. A special branch of the hierarchy called *Global* displays global functions.

You can also use the Class View window to view function code and set breakpoints for a class.

Displaying the Class View window

To open the Class View window:

1. Select **View Classes** from the C++ menu, or click on the **View Classes** button in the toolbar. A Class View window is displayed that shows the class hierarchy of your C++ program. Figure 10-2 shows an example of the Class View window.

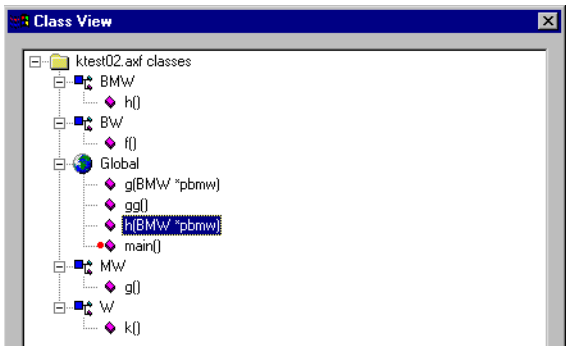


Figure 10-2 Class View window

Viewing code from the Class View window

To view the source code for a class:

1. Display the Class View window.
2. Click the right mouse button on a member function. A Class View window menu is displayed (see Figure 10-3).

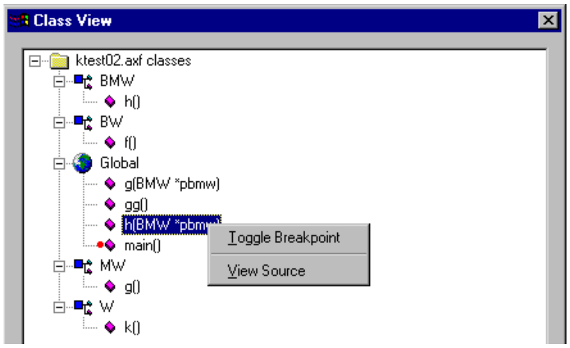


Figure 10-3 Class View window menu

3. Select **View Source** from the Class View window menu to display the source code for the function. You can also double-click the left mouse button on a member function to display the function source.
4. Select **Set or Edit Breakpoint...** from the Execute menu if you want to add a breakpoint within the code you are viewing. Refer to *Setting and clearing breakpoints from the Class View window* on page 10-4 for information on how to set a breakpoint at function entry.

Setting and clearing breakpoints from the Class View window

To toggle a breakpoint in the program when the source for a class or function is entered:

1. Display the Class View window.
2. Click the right mouse button on a member function. A Class View window menu is displayed (see on page 10-4).
3. Select **Toggle Breakpoint** from the Class View window menu to set a breakpoint, or unset an existing breakpoint. Breakpoints are indicated by a red dot to the left of the function in the Class View window.

10.2.2 Using the Watch window

The Watch window allows you to set watches on variables and expressions. It provides similar functionality to the debugger Local and Global windows. In addition, it provides a C++ interpretation of the data being displayed.

Note

The Watch window is *not* used to set watchpoints. Select **Set or Edit Watchpoint...** from the Execute menu to set watchpoints. Refer to *Watchpoints* on page 9-7 for more information.

Evaluation of function pointers and member functions is not available in this version of ADW or ADU.

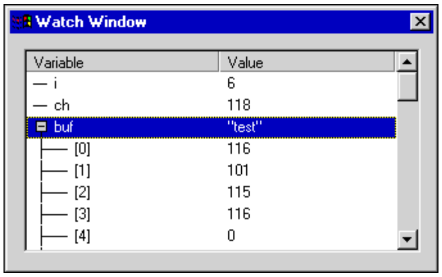
You can specify the contents and format of the Watch window using the Watch window menu. The following sections describe how to:

- view the Watch window
- display the Watch window menu
- delete and add watch items
- format watch items
- change the contents of watched items
- recalculate watches.

Viewing the Watch window

To view the Watch window:

1. Select **Show Watch Window** from the C++ menu or click on the **Show Watches** button in the toolbar. The Watch window displays a list of watched variables and expressions. Figure 10-4 shows an example.


Watch Window

Variable	Value
i	6
ch	118
buf	"test"
[0]	116
[1]	101
[2]	115
[3]	116
[4]	0

Figure 10-4 Watch window

Expressions that return a scalar value are displayed as an expression-value pair. Non-scalar values, such as structures and classes, are displayed as a tree of member variables. If a class is derived, the base classes are represented by `::<base class>` member variables of the class.

Note

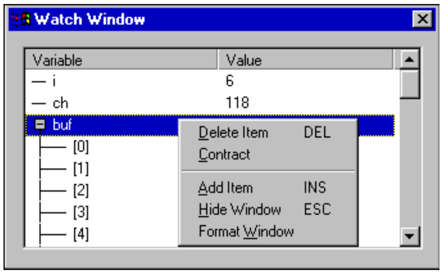
You can also open the Watch window from the Evaluate Expression window. Refer to *Evaluating expressions and adding watches* on page 10-10 for more information.

Displaying the Watch window menu

The Watch window menu enables you to add and delete watches, to change the display format of watches, and to change the contents of watched variables. To display the Watch window menu:

1. Display the Watch window.
2. Click the right mouse button in the Watch window. The Watch window menu is displayed. This menu is context-sensitive. The menu items displayed depend on:
   - whether or not you have clicked on an existing watch item
   - the type of watch item you have clicked on.

   For example, Figure 10-5 shows the Watch window menu that is displayed when the right mouse button is clicked on the character array `buf`.



Variable	Value
i	6
ch	118
buf	
[0]	
[1]	
[2]	
[3]	
[4]	

Figure 10-5 Watch window menu

Deleting a watch item

To delete a watch item from the Watch window:

1. Display the Watch window.
2. Either:
   - click the right mouse button on the item you want to delete and select **Delete Item** from the Watch window menu
   - click on the item you want to delete and press the Delete key.

   The watch item is deleted from the Watch window.

Adding a watch item

To add a watch item to the Watch window:

1. Display the Watch window.
2. Either:
   - click the right mouse button in the Watch window to display the Watch window menu and select **Add Item** from the Watch window menu
   - press the Insert key.

   A Watch Control window is displayed (see Figure 10-6).

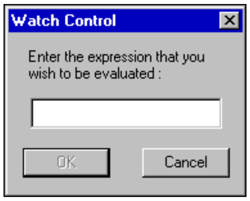


Figure 10-6 Watch Control window

3. Enter an expression to add to the Watch window and click **OK**. Refer to *Evaluating expressions and adding watches* on page 10-10 for more information on the types of expression you can add to the Watch window.

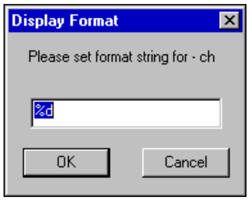
——— **Note** ———

You can also add an expression to the Watch window directly from the Evaluate Expression window. Refer to *Evaluating expressions and adding watches* on page 10-10 for more information.

Formatting watch items

To change the formatting of values displayed in the Watch window:

1. Display the Watch window.
2. Right-click in the Watch window to display the Watch window menu.
3. Select **Format Window** to format all items in the window. The Display Format window is displayed (Figure 10-7).



Please set format string for - ch

%d

Figure 10-7 Display Format window

4. Enter a format string for the item, or items in the window. You can enter any single print conversion specifier that is acceptable as an argument to ANSI C `sprintf()` as a format string, except that `*` cannot be used as a precision. For example, enter `%x` to format values in hexadecimal, or `%f` to format values as a character string (see also *Working with variables* on page 9-13).
5. Click **OK** to apply the format change.

Changing the contents of watched items

To change the contents of items in the Watch window:

1. Display the Watch window.
2. Display the Modify Item window (see Figure 10-8) by double-clicking on the item you want to change. Alternatively, right-click on the item you want to change and select **Edit value** from the Watch window menu.

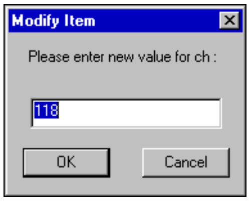


Figure 10-8 Modify Item window

3. Enter a new value for the variable.
4. Click **OK** to change the contents of the variable.

Recalculating watches

Select **Recalculate Watches** from the C++ menu or click on the **Recalculate Watches** button in the toolbar to reinitialize the Watch window to its original state, with all structures and classes expanded by one level. You can use this menu item if the value of any variable might have been changed by external hardware while the debugger is not stepping through code.

10.2.3 Evaluating expressions

The Evaluate Expression window allows you to enter a simple C++ expression to be evaluated. The Evaluate Expression window provides similar functionality to the debugger Expression window, with a C++ interpretation of the data being displayed.

Evaluating expressions and adding watches

To enter an expression to be evaluated:

1. Select **Evaluate Expressions** from the C++ menu or click on the **Evaluate Expression** button in the toolbar. The Evaluate Expression window is displayed (Figure 10-9).

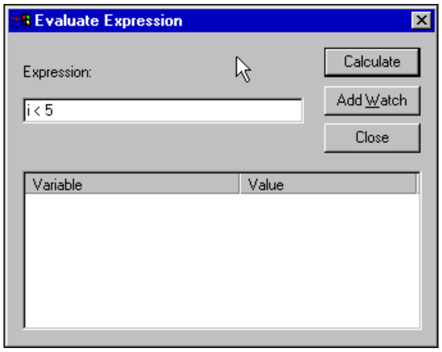


Figure 10-9 Evaluate Expression window

2. Enter the expression to be evaluated and press the Enter key, or click on the **Calculate** button. The value of the expression is displayed:
   - If the expression is a variable, the value of the variable is displayed.
   - If the expression is a logical expression, the window displays `1` if the expression evaluates to true, or `0` if the expression evaluates to false.
   - If the expression is a function, the value of the function is displayed. Member functions of C++ classes cannot be evaluated.

   Refer to *Expression evaluation guidelines* on page 10-11 for more information on expression evaluation in C++.

3. Click on the **Add Watch** button to add the expression to the Watch window.

Expression evaluation guidelines

The following rules apply to expression evaluation for C++:

- You cannot use member functions of C++ classes in expressions.
- You cannot use overloaded functions in expressions.
- You can use only 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 ADW and ADU use the C operator instead. Specific C++ operators, such as the scope operator `::`, are not recognized.
- You cannot access bBase classes in standard C++ notation. For example:

```
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, the expression evaluation window does not accept `this->Base::name` because ADW and ADU do not understand the scope operator. You can access this value with:

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

- Though it is possible to call member functions in the form `Class::Member(...)`, this gives undefined results.
- `private`, `public`, and `protected` attributes are not recognized in ADW or ADU expression evaluation. This means that you can use private and protected member variables during expression evaluation because they are all treated as public.

Part C
armsd

Chapter 11
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:

11.1 About armsd

The *ARM symbolic debugger* (armsd) can be used to debug programs built using the ARM tools.

11.1.1 Selecting a debugger

armsd supports:

- debugging using ARMulator
- remote debugging using ADP.

11.1.2 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`.

11.2 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 13-7.

11.2.1 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.

`-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.

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=n,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 13-48 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 13-48 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 13-28 for more information.

`args` Gives program arguments. You can also specify this information using the `load` command. See *load* on page 13-28 for more information.

Chapter 12
Getting Started in armsd

This chapter includes further information about the use of the *ARM Symbolic Debugger* (armsd). It contains the following sections:

12.1 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* on page 12-2
- *Variable names and context* on page 12-2
- *Program locations* on page 12-4
- *Expressions* on page 12-5
- *Constants* on page 12-6.

12.1.1 Command syntax conventions

The following conventions are used in descriptions of armsd commands:

`typewriter`	Shows command elements that you should type at the keyboard.
<u>`type`</u>`writer`	Many command names can be abbreviated. Underlined text shows the permitted abbreviation of a command.
`typewriter`	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.

12.1.2 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
```

Global variables

You can access global variables by qualifying them with the module 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 context is:

```
{#}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.

12.1.3 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 *Low-level symbols* on page 12-13 for further details.

Procedure name

Using a procedure name alone sets a breakpoint (see *break* on page 13-12) 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:123
procedure: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 third statement in line 100.

12.1.4 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 12-1, in descending order of precedence.

Table 12-1 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`

Table 12-1 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.

12.1.5 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 `@`.

12.2 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* on page 12-7
- *Accessing variables* on page 12-10
- *Formatting printed results* on page 12-11
- *Specifying the base for input of integer constants* on page 12-12.

12.2.1 Summary of armsd variables

You can modify many debugger defaults by setting variables. Table 12-2 lists the variables. Most of these are described elsewhere in this chapter in more detail.

Table 12-2 armsd variables

Variable	Description
`$clock` (ARMulator only)	Number of microseconds since simulation started. This read-only variable is available only if a processor clock speed is specified. See *ARMulator configuration* on page 5-82 for information on specifying the simulated processor clock speed.
`$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”).
`$inputbase`	Base for input of integer constants (initially 10).
`$list_lines`	Default number of lines for list command (initially 16).

Table 12-2 armsd variables (continued)

Variable	Description
`$fpresult`	Floating-point value returned by last called function (junk if none, or if a floating-point value was not returned). A read-only variable. `$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 5-82 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.
`$vector_catch`	Indicates whether or not execution is interrupted when various exceptions occur. The default value is `%RUsPDAifE`. Capital letters indicate that the exception is to be intercepted: `R` Reset `U` Undefined Instruction `S` SWI `P` Prefetch Abort `D` Data Abort `A` Reserved (do not use) `I` IRQ `F` FIQ `E` Reserved (do not use)
`$rdi_log`	RDI logging is enabled if nonzero, and serial line logging is enabled if bit 1 is set (initially 0).
`$top_of_memory`	This variable informs the debugger where the top of RAM is on your target. 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.

Table 12-2 armsd variables (continued)

Variable	Description
`$target_fpu`	This variable controls the way that floating-point values are interpreted by the debugger. It is important for correct display of float and double values in memory that this variable is set to a value that is appropriate for the target in use. If you attempt to change this value, a validity test ensures that the only settings allowed are those that are compatible with the representation of floating-point values in the current image. Valid settings and their meanings are: **1** Selects pure-endian doubles (`softVFP`). This is the default setting for images built with ADS tools. Values are read from ordinary registers. **2** Selects mixed-endian doubles (`softFPA`). Values are read from ordinary registers. **3** Selects hardware Vector Floating-Point unit (`VFP`). Values are read from registers CP10 and CP11. **4** Selects hardware Floating-Point Accelerator (`FPA`). Values are read from registers CP1 and CP2. **5** Reserved. All other values are invalid and result in `$target_fpu` being set to 1. 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 Compiler, Linker, and Utilities Guide*.

`$sourcedir`	This variable contains a list of the paths to be searched when a source file is required. It defaults to NULL if no value is specified. When you specify search paths:
	• Enclose the full pathname in double quotes.
	• In ADW and armsd under Windows DOS, escape the backslash directory separator with another backslash character. For example:
	`$sourcedir="c:\\mysource\\src1"`
	• Separate multiple pathnames with a semicolon, not with a space character. For example:
	`$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.

armsd internal variables

The variables in Table 12-3 are included to support EmbeddedICE.

Table 12-3 armsd variables for Multi-ICE and EmbeddedICE

Variable	Description
`$icebreaker_lockedpoints`	Shows or sets locked EmbeddedICE logic points.
`$semihosting_enabled`	Enables or disables semihosting.
`$semihosting_vector`	Sets up semihosting SWI vector (described in the *ADS Debug Target Guide*).
`$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.

12.2.2 Accessing variables

The following commands are available for accessing variables.

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 allows 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).

12.2.3 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`
- `$float_format`
- `$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%.8lx`" 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 12-7.

12.2.4 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 12-11.

12.3 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.

12.3.1 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 `@`.

Note

Low-level symbols do not have a context and so they are always available.

12.3.2 Predefined symbols

There are several predefined symbols, as shown in Table 12-4. To differentiate these from any high-level or low-level symbols in the debugging tables, precede them with #.

Table 12-4 High-level symbols for low-level entities

Symbol	Description
r0 - 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).

Table 12-4 High-level symbols for low-level entities (continued)

Symbol	Description
`sb`	This is the ATPCS static base in RWPI variants of the ATPCS (r9/v6).
`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).

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
```

For example, use `#pc=0` instead of `pc=0`.

Chapter 13
Working with armsd

This chapter lists and explains every command supported by the *ARM Symbolic Debugger* (armsd). It contains the following sections:

13.1 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 13-7.

The functional groups are:

- *Symbols* on page 13-2
- *Controlling execution* on page 13-3
- *Reading and writing memory* on page 13-3
- *Program context* on page 13-3
- *Low-level debugging* on page 13-4
- *Coprocessor support* on page 13-4
- *Profiling commands* on page 13-5
- *Miscellaneous commands* on page 13-5.

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.

———————————

13.1.1 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.

13.1.2 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.
`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.

13.1.3 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.

13.1.4 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.

13.1.5 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.
`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.

13.1.6 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 *CoProcessor Data Transfer* (CPDT) or a *CoProcessor Register Transfer* (CPRT) 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.

13.1.7 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.

13.1.8 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 allows 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.

13.1.9 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 13-46.

13.1.10 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 13-47.

13.2 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 13-9.

13.2.1 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 12-2.
Expression	An arbitrary expression using the constants, variables, and operators described in *Expressions* on page 12-5. 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 12-4).
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 12-2 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.
- A C `printf()` function format descriptor. Table 13-1 shows some common descriptors.

Table 13-1 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.

13.2.2 ! 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.

13.2.3 | command

The `|` command introduces a comment line.

Syntax

The syntax of `|` is:

`|`*`comment`*

where:

`comment`	Is a text string.

Usage

This command allows you to annotate your armsd script file.

13.2.4 alias

The `alias` command defines, undefines, or lists aliases. It allows 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”
```

13.2.5 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.

13.2.6 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.

13.2.7 break

The `break` command allows 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 12-4.
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 13-42.

13.2.8 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.

13.2.9 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}*}*
```

where:

cpnum	Identifies the coprocessor.
rno{:*rno1*}	Identifies the register set.
size	Is the register size in bits.
access	Can comprise the letters: `R` The register is readable. `W` The register is writable. `D` The register is accessed through `CPDT` instructions (if not present, the register is accessed through `CPRT`s).

values	The format depends on whether the register is to be accessed through `CPRT` instructions. If so, 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 not, it comprises two integer values to form bits 12 to 15 and bit 22 of `CPDT` instructions to read and write the register:
	`b12_15, b22`

displaydesc	Describes how the contents of the registers are to be formatted for display, and takes one of the forms listed in Table 13-2 on page 13-15.

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:

```
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 13-2 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*]. There must be one string for each possible value of `field`. 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. There must be one character in *letters* for each bit of `field`. The letters are displayed in uppercase if the corresponding bit of the field is set, and in lowercase if it is clear. The first letter represents the lowest bit if *bit1* < *bit2*, otherwise it represents the highest bit.	

13.2.10 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.

13.2.11 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 13-2 on page 13-15.

13.2.12 cregdef

The `cregdef` command describes how the contents of a coprocessor register are formatted for display.

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 13-2 on page 13-15.

13.2.13 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 13-14.

13.2.14 examine

The `examine` command allows you to examine the contents of memory.

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 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. Low-level symbols are accepted by default.

13.2.15 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
```

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.

13.2.16 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
```

`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
`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.

13.2.17 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.

13.2.18 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.

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
```

13.2.19 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 *`.

13.2.20 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.

13.2.21 istep

The `istep` command steps execution through one or more instructions.

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.

13.2.22 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.

13.2.23 let

The `let` command allows 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 12-11 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
```

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:

```
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.

13.2.24 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.

13.2.25 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 11-3 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 12-2 on page 12-7.

13.2.26 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).

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
```

13.2.27 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.

13.2.28 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.

13.2.29 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.

Usage

You can store frequently-used command sequences in files, and call them using `obey`.

13.2.30 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.

13.2.31 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.

13.2.32 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 13-1 on page 13-8. 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 12-10 for more information on the `print` command.

13.2.33 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 Compiler, Linker, and Utilities Guide*.

13.2.34 profoff

The `profoff` command stops the collection of profiling data.

Syntax

The syntax of the `profoff` command is:

```
profoff
```

Usage

For more information on the ARM profiler, refer to the *ADS Compiler, Linker, and Utilities Guide*.

13.2.35 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.

13.2.36 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.

Usage

The generated information can be viewed using the `armprof` utility. This is described in the *ADS Compiler, Linker, and Utilities Guide*.

13.2.37 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* is preceded 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
```

13.2.38 quit

The `quit` command terminates the current armsd session.

Syntax

The syntax of the `quit` command is:

```
quit
```

Usage

This command also closes any open log or obey files.

13.2.39 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).

13.2.40 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 12-14.

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.

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 13-1

```
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
```

13.2.41 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:

<u>`rel`</u>`oad{/`*`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.	

Usage

Breakpoints (but not watchpoints) remain set after a `reload` command.

13.2.42 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.

13.2.43 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 source object 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).

13.2.44 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.

13.2.45 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.

13.2.46 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.

13.2.47 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 13-40.

13.2.48 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.

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.

13.2.49 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.

13.2.50 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).

13.3 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*.

13.3.1 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.

13.3.2 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.

13.4 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.

13.4.1 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.

13.4.2 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.

13.4.3 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 13-48.

Syntax

The syntax of the `loadconfig` command is:

```
loadconfig file
```

where:

`file`	Names the EmbeddedICE configuration data file to load.

13.4.4 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 13-48).

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.

Appendix A
AXD and ADW/ADU/armsd Commands

This appendix compares the commands supported by the command-line interface of AXD with those supported by ADW, ADU, and 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.

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.

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 (see Chapter 13)	AXD commands (see Chapter 6)	Short form
!*command*	-	-
|*comment*	comment *string*	com
alias [*name*[*expansion*]]	-	-
arguments [*context*]	-	-
backtrace [*count*]	backtrace[*count*] bactrace is an alias of stackentries	 stk
break[/*size*][*loc*[*count*] [do {*command*[; *command*]}] [if *expr*]]	break[*expr*|*position* [*nth_time*]]	br
call[/*size*] *loc*[*expr-list*]	-	-
-	cclasses *class*	ccl
-	cfunctions *class*	cfu
-	classes[*image*]	cl
-	clear	clr
-	clearbreak *breakpoint*|all clearbreak has the alias unbreak	cbr
-	clearstat *referencepoint*	cstat
-	clearwatch *watchpoint*|all clearwatch has the alias unwatch	cwpt

Table A-1 armsd and AXD commands (continued)

armsd commands (see Chapter 13)	AXD commands (see Chapter 6)	Short form
coproc cpnum[rno[:rno1] size access values [displaydesc]*]*	-	-
context context	context[context]	con
-	convariables[context] [scope][format]	convar
cregisters cpnum	-	-
cregdef cpnum rno displaydesc	-	-
-	cvariables class	cva
cwrite cpnum rno val [val ...]*	-	-
-	dbginternals	di
-	disassemble expr1 [+]expr2[asm] disassemble has the alias list	dis
let $echo 0|1	echo toggle	-
examine[expr1] [, [+]expr2]	examine expr1, [+]expr2 [memory[format]] 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
fpregisters[/full]	-	-
-	functions[image]	fu
getfile filename expression	getfile file addrexpr getfile is an alias of loadbinary	lb

Table A-1 armsd and AXD commands (continued)

armsd commands (see Chapter 13)	AXD commands (see Chapter 6)	Short form
go[while *expression*]	go[*processor*] 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* let is an alias of setwatch	swat
list[/*size*][*expr1*] [, [+]*expr2*]	list *expr1* [+]*expr2*[*asm*] list is an alias of disassemble	dis
-	listformat[*nbits*]	lsfmt
load[/*profile-opt*] *image-file*[*args*]	load *file*[*processor*]	ld
-	loadbinary *file addrexpr* loadbinary has the alias getfile	lb
-	loadsession *sesfile*	lss
-	loadsymbols *file*[*processor*] loadsymbols has the alias readsyms	lds
localvar *vartype varname*	-	-
log *filename*	log[*file*]	-
lsym *pattern*	lowlevel[*image*]	lsym

Table A-1 armsd and AXD commands (continued)

armsd commands (see Chapter 13)	AXD commands (see Chapter 6)	Short form

-	`memory` *`expr1`* `[+]`*`expr2`*
	`[` *`memory`*`[` *`format`*`]]`
	`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`*`]`
	`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`*
	`putfile` is an alias of `savebinary`
		
		`sb`

`quit`	`quitdebugger`	`quitd`

`readsyms` *`filename`*	`readsyms` *`file`*`[` *`processor`*`]`
	`readsyms` is an alias of `loadsymbols`
		
		`lds`

-	`record[` *`file`*`]`	`rec`
-	`regbanks[` *`processor`*`]`	`regbk`
-	`registers[` *`regbank`*`[` *`format`*`]]`	`reg`
`reload[/`*`profile-option`*`] [` *`arguments`*`]`	`reload[` *`image`*`]`	`rld`

Table A-1 armsd and AXD commands (continued)

armsd commands (see Chapter 13)	AXD commands (see Chapter 6)	Short form

-	`run[ `*`processor`*`]`
	`run` has the alias `go`
		`r`

-	`runtopos `*`position`*`[ `*`processor`*`]`	`rto`

-	`savebinary `*`file expr1`*` [+]`*`expr2`*
	`savebinary` has the alias `putfile`
		`sb`

-	`savesession `*`sesfile`*	`ss`
-	`setbreakprops `*`breakpoint propid value`*	-
-	`setimgprop `*`image ipvar value`*	`sip`

-	`setmem `*`addrexpr valexpr`*
	`[ `*`memory`*`]`
		`smem`

`pc=xx`	`setpc `*`expr`*	`pc`
rn=xx	`setproc `*`processor`*	`sproc`
-	`setprocprop `*`ppvar value`*	`spp`
-	`setreg [`*`regbank`*`|]`*`register expr`*	`sreg`
-	`setsourcedir `*`directory_list`*	`ssd`

-	`setwatch `*`expr1`*`, `*`expr2`*
	`setwatch` has the alias `let`
		`swat`

-	`setwatchprops `*`watchpoint propid value`*	`swp`

-	`source `*`value1`*` [+]`*`value2`*` [ `*`file`*`]`
	`source` has the alias `type`
		`src`

-	`sourcedir[ `*`path`*`[  `*`index`*`]]`	`sdir`

`ba`cktrace [*count*]	`stackentries[ `*`count`*`]`
	`stackentries` has the alias `backtrace`
		`stk`

`in`	`stackin`	`in`
`out`	`stackout`	`out`

Table A-1 armsd and AXD commands (continued)

armsd commands (see Chapter 13)	AXD commands (see Chapter 6)	Short form
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*] type is an alias of source	src
unbreak [*location*|*#breakpoint_num*]	unbreak *breakpoint* unbreak is an alias of clearbreak	cbr
unwatch [*variable*|*#watchpoint_num*]	unwatch *watchpoint* unwatch is an alias of clearwatch	cwpt
variable *variable*	variables[*image*]	va
print *expr*	watch *expr*[*format*] watch has the alias print	wat
watch[*variable*]	watchpt[*expr*[*nth_time*]]	wpt
where[*context*]	where[*context*]	-
statement;[*statement*;] while *expr*	-	-

A.2 Useful internal variables

Table A-2 lists some variables with values that you might want to examine or change.

In ADW, ADU, or armsd, you examine these as debugger internal variables and can change their values with a let command. In AXD more CLI commands are available.

Table A-2 Internal variables

ADW/ADU/armsd variable	AXD command (see Chapter 6)
`$vector_catch`	`pp` to examine, `spp` to change
`$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.
`$user_input_bit1`	Hardware input to Multi-ICE only. Not writeable.
`$user_input_bit2`	Hardware input to Multi-ICE only. Not writeable.
`$user_output_bit1`	`pr` to examine, `let` to change
`$user_outout_bit2`	`pr` to examine, `let` to change
$arm9_restart_code_address	`pr` to examine, `let` to change. Multi-ICE 1.3 and 1.4

Table A-2 Internal variables (continued)

ADW/ADU/armsd variable	AXD command (see Chapter 6)
$cache_clean_code_address	`pr` to examine, `let` to change. Multi-ICE 2.0.
`$sw_breakpoints_preferred`	`pr` to examine, `let` to change. Multi-ICE only.
`$sourcedir`	`sdir` to examine, `ssd` to change
`$echo`	echo on|off

Appendix B
Coprocessor Registers

This appendix describes coprocessor registers for various ARM processors. It contains the following sections:

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

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

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

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
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

Table B-4 ARM920T rev 0 (continued)

Name	Description	Register
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

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
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

Table B-5 ARM920T rev 1 (continued)

Name	Description	Register
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

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

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

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

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

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
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

Table B-8 ARM946E-S (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: 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

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

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
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

Table B-10 ARM10200E (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
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
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

Table B-10 ARM10200E (continued)

Name	Description	Register
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
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

Table B-10 ARM10200E (continued)

Name	Description	Register
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

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
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

Table B-11 ARM1020E (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

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
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

Table B-12 ARM10E (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

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
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

Table B-13 XScale (continued)

Name	Description	Register
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
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

Table B-13 XScale (continued)

Name	Description	Register
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
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

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*.
ADU	See *ARM Debugger for UNIX*.
ADW	See *ARM Debugger for Windows*.
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 Debugger for UNIX	ARM Debugger for UNIX (ADU) is the UNIX version of the ARM Debugger for Windows. This debugger will not be supported in future versions of the ARM Developer Suite.
ARM Debugger for Windows	ARM Debugger for Windows (ADW). This debugger will not be supported in future versions of the ARM Developer Suite.
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.
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 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.
ARMulator	ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
armsd	See *ARM Symbolic Debugger*.
ATPCS	ARM/Thumb Procedure Call Standard.
AXD	See *ARM eXtended Debugger*.
Backtracing	See *Stack backtracing* and *Tracing*.
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*.
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 ADW, ADU and AXD all offer 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.
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.
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*.
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 may also be an image file or a session file.
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*.)
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	CodeWarrior is an example of an IDE, offering facilities for automating image-building and file-management processes.
Joint Test Access Group	Many debug and programming tools use a *Joint Test Access Group* (JTAG) interface port to communicate with processors. For further information refer to IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).
JTAG	See *Joint Test Access 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.
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.

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 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.

Program image

See *Image*.

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 open ARM standard procedural interface between a debugger and the debug agent. The widest possible adoption of this standard is encouraged. 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*.

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).
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 may consist of a single, simple GUI control such as an edit field or a more complex multi-control dialog implemented as an ActiveX. 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.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.