ARM Developer Suite

Version 1.0.1

Debuggers Guide

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Change
October 1999	A	Release 1.0
February 2000	B	Release 1.0.1

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 Preface-viii
- *Feedback* on page Preface-xii.

About this book

This book has three main parts in which all the currently supported ARM debuggers are described:

- Part A describes the graphical user interface components of *ARM eXtended Debugger* (AXD), the most recent ARM debugger and part of the ARM Developer Suite of software. 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 need to refer to the Glossary and Index near 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 Window NT, 95, or 98, 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 Preface-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 need, 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 ARM 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, which is an interactive, command-line, source-level debugger providing high-level debugging support for languages such as C, and low-level support for 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 *Debug Communications Channel*
This appendix explains how to use the ARM Debug Communications Channel. You can use this feature from any ARM debugger.

Glossary An alphabetically arranged glossary defines the special terms used.

Index A comprehensive alphabetical index completes this book.

Typographical conventions

The following typographical conventions are used in this book:

`typewriter` Denotes text that may be entered at the keyboard, such as commands, file and program names, and source code.

`typewriter` (with `type` underlined) Denotes a permitted abbreviation for a command or option. The underlined text may 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 that is specific to the ARM debuggers supplied with the *ARM Developer Suite* (ADS). Refer to the following books in the ADS document suite for information on other components of ADS:

- *Getting Started* (ARM DUI 0064A)
- *ADS Tools Guide* (ARM DUI 0067A)
- *CodeWarrior IDE Guide* (ARM DUI 0065)
- *ADS Debug Target Guide* (ARM DUI 0058A)
- *ADS Developer Guide* (ARM DUI 0056A).

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

- *ARM Architecture Reference Manual* (ARM DUI 0100). This is supplied in Dynatext format, and in PDF format in `install_directory\PDF\ARM-DDI0100B_armarm.pdf`.
- *ARM Applications Library Programmer's Guide* (ARM DUI 0081). This is supplied in Dynatext format, and in PDF format on the CD.
- *ARM ELF specification* (SWS ESPC 0003). This is supplied in PDF format in `install_directory\PDF\specs\ARM ELFA08.pdf`.
- *TIS DWARF 2 specification*. This is supplied in PDF format in `install_directory\PDF\specs\TIS-DWARF2.pdf`.
- *Angel Debug Protocol*. This is supplied in PDF format in `install_directory\PDF\specs\ADP ARM-DUI0052C.pdf`
- *Angel Debug Protocol Messages*. This is supplied in PDF format in `install_directory\PDF\specs\ADP ARM-DUI0053D.pdf`

In addition, refer to the following documentation 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.

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 number of the tool, including the version number and build number.

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.

This chapter contains the following sections:

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 being emulated by software. Even though you might run such software on the same computer as the debugger, it is always helpful to think of the target as being a separate piece of hardware.

An alternative prototype product might be built on a printed circuit board, and include one or more processors, on which you can run and debug software.

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

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 such 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, such as:

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

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

Examples of debug agents include:

- Multi-ICE
- EmbeddedICE

- ARMulator
- BATS
- Angel.

Multi-ICE and EmbeddedICE are separate products. They are 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 1-5). 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)
- 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 1-6.

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, that 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:

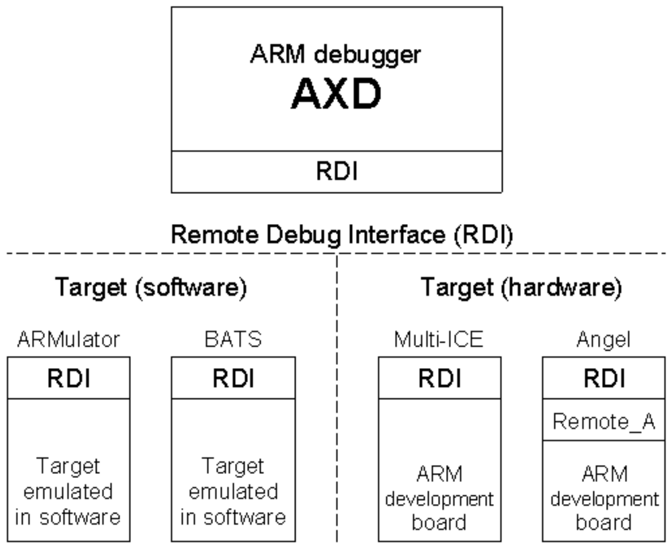


Figure 1-1 Debugger-target interface

An ARM Development Board, communicating through Multi-ICE, or Angel, is an example of a hardware target system. ARMulator and BATS are examples 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*, which can be:
  - — hardware-based on an ARM core
  - — software that emulates an ARM core.
- a *debugger,* such as AXD, ADW, ADU, or armsd.

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

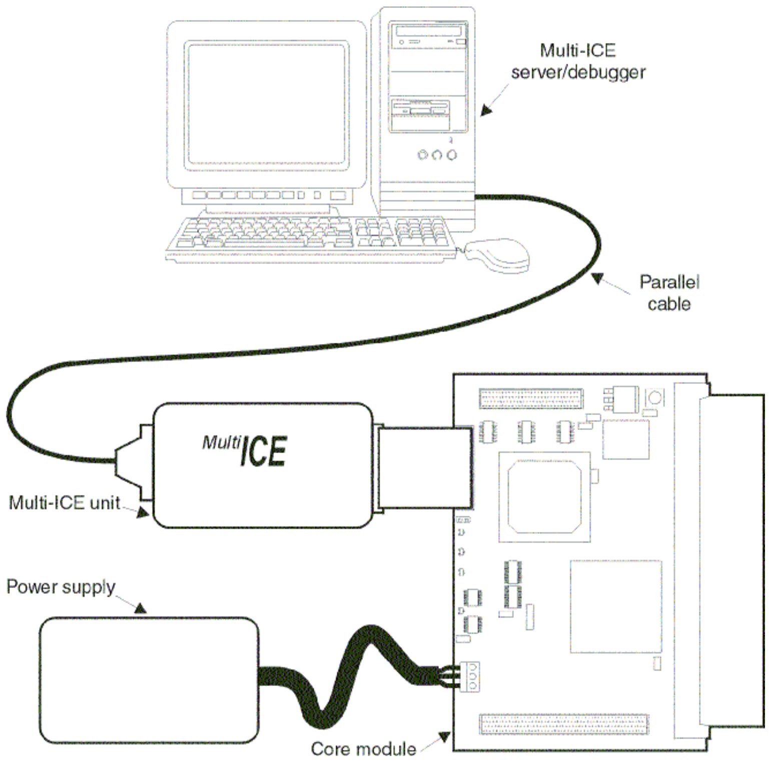


Figure 1-2 A typical debugging set-up

1.2.2 Debugging systems

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

- *ARMulator* on page 1-7
- *Basic ARM Ten System (BATS)* on page 1-7
- *Multi-ICE and EmbeddedICE* on page 1-7
- *Angel debug monitor* on page 1-8.

See *Configure Target...* on page 5-56 for information about the configuration of debugger target systems.

ARMulator

ARMulator is a collection of programs that emulate 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. See the *ADS Debug Target Guide* for more information about ARMulator.

Basic ARM Ten System (BATS)

For systems based on ARM10 processors, the *Basic ARM Ten System* (BATS) models transactions which take place over buses connecting processors, coprocessors, switches, and memory. You can select combinations of processors, coprocessors, switches, and memory to model your proposed hardware. You can also write your own modules, or copy and edit existing ones.

BATS models transactions in detail. You can investigate bus contention between multiple bus masters, and measure benchmark timings accurately. See the *ADS Debug Target Guide* for more information about BATS.

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 services of the host system, such as screen display, keyboard input, and disk drive storage by means of semihosting.

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

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 by means of 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.

1.2.3 Availability and compatibility

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

The ARM Developer Suite (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, the latest ARM debugger, is recommended. However, if you have used the earlier ARM Software Development Toolkit, then you might prefer to use the older ADW, ADU, or armsd debuggers because they offer all the facilities you need in a way that is familiar to you.

The earlier debuggers are included on the CD ROM so that during installation you can replace the entire existing ARM Software Development Toolkit with the new ARM Developer Suite and still have all the supported debuggers.

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

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

A C++ compiler is supplied as part of ADS, and is no longer an extra-cost option as was the case in the past.

1.3 Online help

Online help is intended to complement 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.

1.3.1 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.

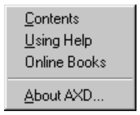


Figure 1-3 Help menu

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 on-screen 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. If this option is not yet available, you can select **Start** → **Programs** → **ARM Developer Suite v1.0** → **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 usually obtain it quickly by applying for it by email or fax.

You can use certain license-managed software without a 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 *Getting Started* book.

2.2 Starting and closing AXD

This section describes how to start and close AXD.

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 v1.0** → **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, 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 DOS, optionally with arguments (see *AXD arguments*).

2.2.2 AXD arguments

The possible arguments (which must be in lower case) for AXD are:

`-debug` *`ImageName`*	Load *`ImageName`* for debugging.
`-exec` *`ImageName`*	Load and run *`ImageName`*.
`-logo`	Show splash screen (this is the default).
`-nologo`	Suppress splash screen.
`-script` *`ScriptName`*	Obey the *`ScriptName`* on startup. This is the equivalent of typing `obey` *`ScriptName`* as soon as the debugger starts up.

For example, to launch AXD and load `sorts.axf` for debugging, type:

```
axd -debug sorts.axf
```

Where an AXD command line includes an image name any options following the image name will be taken as command line options for the image being loaded, not AXD.

2.2.3 Closing AXD

To close down AXD, select **Exit** from the **File** menu or click the **X** button at the far right of the AXD title bar.

2.3 Debugger target

This section explains how to set up the target hardware, or emulator, on which to run the software to be debugged, using:

- *ARMulator*
- *BATS* on page 2-5
- *Multi-ICE unit and target board* on page 2-5
- *Angel or EmbeddedICE* on page 2-6.

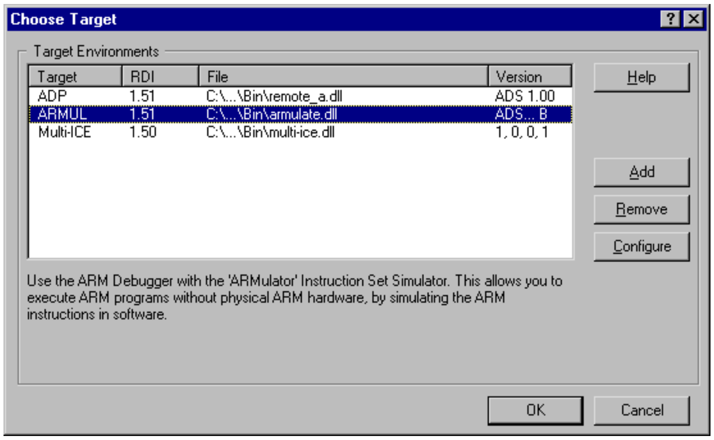
The first time you run AXD, ARMulator is selected by default as the target, with default settings taken from a configuration file. Subsequently, AXD starts up with the last used target configuration still effective.

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 debugger:

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	ADS 1.00
ARMUL	1.51	C:\...\Bin\armulate.dll	ADS... B
Multi-ICE	1.50	C:\...\Bin\multi-ice.dll	1, 0, 0, 1

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 5-56.

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 BATS

To start a BATS 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 2-4.
2. Select the BATS target. If BATS is not yet in the list of available target environments, click **Add**, locate and select `bats.dll`, click **Open**, and BATS is added to the list and selected.
3. 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 5-56.
4. When you have selected BATS as the target, and configured it if necessary, click **OK**. You can now load an image onto the target and control its execution.

2.3.3 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 2-4.
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 5-56.
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.

2.3.4 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 2-4.
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 5-56.
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.

2.3.5 Gateway DLL

To target the Gateway DLL:

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 2-4.
2. Click **Add…**. A standard file dialog is displayed (Figure 2)
3. Select `gateway.dll` and click **Open**.
4. Click **Configure…** in the Choose Target screen. The Gateway Remote Configuration panel is displayed (Figure 2-2 on page 2-7).

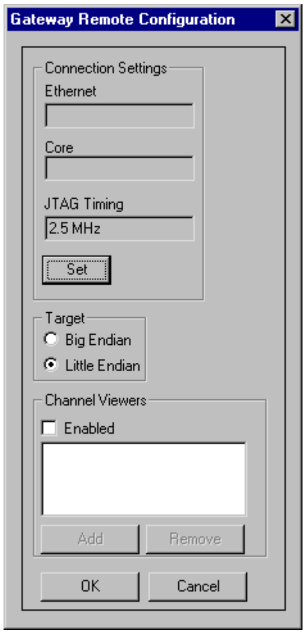


Figure 2-2 Gateway Remote Configuration panel

5. Click **Set** to set connection details for your probe. A Set Connection Details panel is displayed (Figure 2-3 on page 2-7).

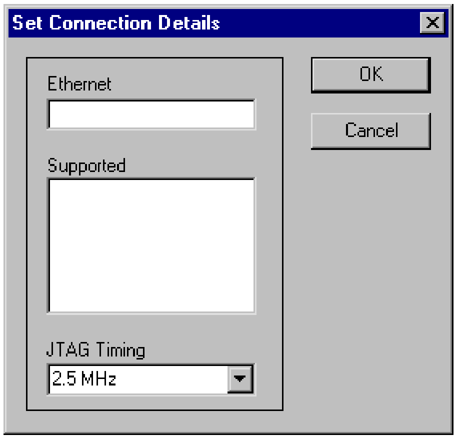


Figure 2-3 Set Connection Details panel

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

7. Press the Tab key, or click in the Supported field. The Debugger establishes a connection with probe and displays a list of supported targets in the Supported field.
8. Select the target type you want in the Supported field.
9. Select the JTAG base clock speed you require from the JTAG Timing 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.
10. Click **OK** to confirm your settings and close the Set Connection Details panel.
11. Click **OK** in the Gateway Remote Configuration panel.
12. 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*
- *Multi-document interface*
- *Docked and floating windows*
- *Tabbed pages* on page 2-10
- *Dialogs* on page 2-10.

2.4.1 Views

A variety of *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 5-13.
- 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 5-33.

2.4.2 Multi-document interface

AXD uses the Windows *Multi-Document Interface* (MDI) so that you can display several windows at the same time. You typically display several processor views and system views. You can arrange your windows in various ways so that, for example, some are docked, some are free-floating, and the remainder are cascaded or tiled.

2.4.3 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, which 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.

2.4.4 Tabbed pages

Several AXD dialogs and property sheets make use of 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 need while making settings or entering data.

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

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

2.4.5 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 its equivalent). 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 2-11.

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 both the target system and the debugger user interface, and enable or disable the display of the status bar and the collection of profiling information.
Window	Allows you to control how MDI windows and icons are displayed.
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 its own pop-up menu offering further items depending on circumstances.

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 the various toolbars, the status bar, keyboard shortcuts, in-place editing, and the command-line interface.

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.

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 5-4).

You can display or hide the status bar (see *Status Bar* on page 5-63).

2.6.4 Keyboard shortcuts

Several kinds of keyboard shortcuts are described in *AXD menus* on page 2-11.

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.

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 and it 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. Chapter 5 *AXD Desktop* gives further details of specific features.

The examples given in this chapter have all been tested and shown to work as described. You may find it useful to follow all the instructions, as a tutorial. Your hardware and software may not be the same as those used for testing these examples, so it is possible that certain addresses or values may 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.

This chapter contains the following sections:

3.1 Running a demonstration program

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

You are likely to be using software such as ARMulator to emulate a debugger target. Alternatively, your target might consist of a Multi-ICE hardware 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 2-5. In any case, you must have selected the target you intend to use and configured it, as described in *Configure Target...* on page 5-56.

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

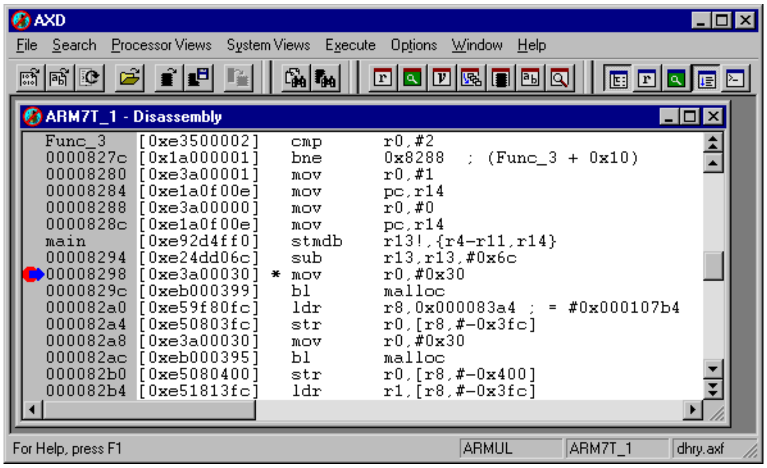
1. Create an executable image by compiling the source code files in the `Dhry` subdirectory and linking the resulting objects with the libraries that they use. If you are running under Windows you can use the CodeWarrior IDE project file `dhry.mcp` supplied. This organizes your work into projects and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

   A blue arrow indicates the current execution point.


AXD
File Search Processor Views System Views Execute Options Window Help
ARM7T_1 - Disassembly
For Help, press F1
ARMUL
ARM7T_1
dhry.axf

```
Func_3    [0xe3500002]   cmp    r0,#2
0000827c  [0x1a000001]   bne    0x8288   ; (Func_3 + 0x10)
00008280  [0xe3a00001]   mov    r0,#1
00008284  [0xe1a0f00e]   mov    pc,r14
00008288  [0xe3a00000]   mov    r0,#0
0000828c  [0xe1a0f00e]   mov    pc,r14
main      [0xe92d4ff0]   stmdb  r13!,{r4-r11,r14}
00008294  [0xe24dd06c]   sub    r13,r13,#0x6c
00008298  [0xe3a00030] * mov    r0,#0x30
0000829c  [0xeb000399]   bl     malloc
000082a0  [0xe59f80fc]   ldr    r8,0x000083a4  ; = #0x000107b4
000082a4  [0xe50803fc]   str    r0,[r8,#-0x3fc]
000082a8  [0xe3a00030]   mov    r0,#0x30
000082ac  [0xeb000395]   bl     malloc
000082b0  [0xe5080400]   str    r0,[r8,#-0x400]
000082b4  [0xe51813fc]   ldr    r1,[r8,#-0x3fc]
```

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 first executable line of source code in function `main()`, where a breakpoint is set by default. A red disc indicates the line where a breakpoint is set,

   A **Source** processor view of the relevant few lines of the relevant file is displayed. Again, a red disc indicates the line where a breakpoint is set, and a blue arrow indicates the current execution point.

4. 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.

5. To repeat the execution of the program, select **Reload Current Image** from the **File** menu, then repeat Steps 3 and 4.

6. 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

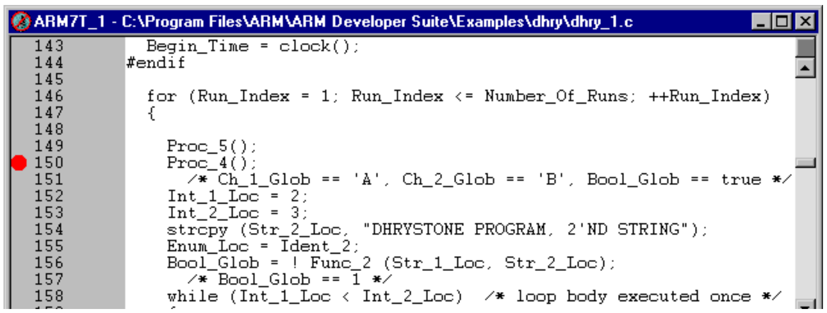
This example runs the same program again, this time with a breakpoint which stops execution a few times. You can examine values when execution stops.

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 first executable line of source code in function `main()` and indicated by a red disc on that line. You can see the source file `dhry_1.c` with a breakpoint and the current position indicated at line number 91.

3. Scroll down through the source file until line number 150 is visible. This is a call of `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**). 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

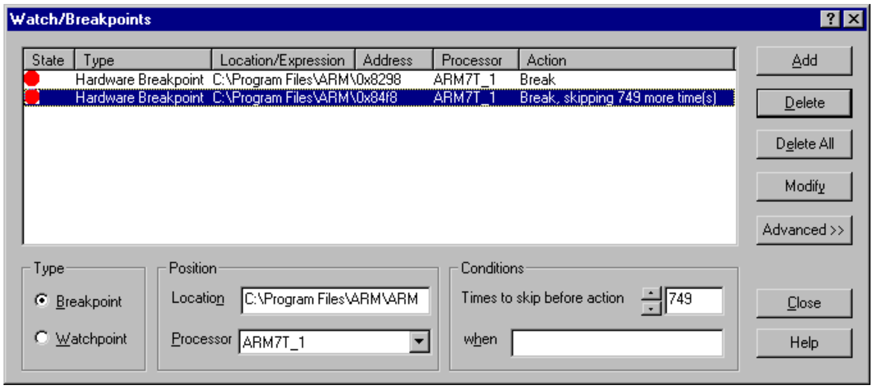
```
143       Begin_Time = clock();
144     #endif
145
146       for (Run_Index = 1; Run_Index <= Number_Of_Runs; ++Run_Index)
147       {
148
149         Proc_5();
150         Proc_4();
151           /* Ch_1_Glob == 'A', Ch_2_Glob == 'B', Bool_Glob == true */
152         Int_1_Loc = 2;
153         Int_2_Loc = 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 */
```

Figure 3-2 Breakpoint set inside loop

5. To edit the details of the new breakpoint, select **Watch/Breakpoints...** from the **Execute** menu.

   Click on the line describing the new breakpoint to select it, and its details appear in the Type, Position, and Conditions boxes. In the **Conditions** box, enter 749 in the **Times to skip before action** field, as shown in Figure 3-3.

   Click **Modify** to update the breakpoint details, then click **Close**.



State	Type	Location/Expression	Address	Processor	Action
●	Hardware Breakpoint	C:\Program Files\ARM\0x8298		ARM7T_1	Break
●	Hardware Breakpoint	C:\Program Files\ARM\0x84f8		ARM7T_1	Break, skipping 749 more time(s)

Type: (● Breakpoint) ○ Watchpoint

Position — Location: C:\Program Files\ARM\ARM; Processor: ARM7T_1

Conditions — Times to skip before action: 749; when:

Figure 3-3 Setting breakpoint details

6. Press **F5** to resume execution, and again enter the number 8000 when prompted. 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 **Formats** → **Dec** 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 7500, 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.)

3.3 Examining the contents of variables

Two methods are described. The first is simpler and shows the contents of specified variables. The second shows the addresses of the variables as well as their contents.

3.3.1 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.

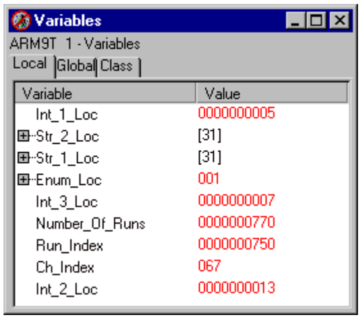
2. Select **Go** from the **Execute** menu (or press **F5**) to reach the first breakpoint, set by default at the first executable line of source code in function `main()`.

3. Select **Variables** from the **Processor Views** menu and 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, and select **Properties...** from the pop-up menu. Select the **Decimal** format option and click **OK**.

4. Press **F5** again, enter a number not much greater than 750 this time for the number of runs required, say 770. The program performs the first 750 executions of the Dhrystone test and stops at the breakpoint you defined in *Setting a breakpoint* on page 3-4. The value of `Run_Index` is displayed in color because it has changed (see Figure 3-4).



Variable	Value
Int_1_Loc	0000000005
⊞ Str_2_Loc	[31]
⊞ Str_1_Loc	[31]
⊞ Enum_Loc	001
Int_3_Loc	0000000007
Number_Of_Runs	0000000770
Run_Index	0000000750
Ch_Index	067
Int_2_Loc	0000000013

Figure 3-4 Examining the contents of variables

5. 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 color.

6. 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.

7. 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.

3.3.2 Addresses and contents of variables

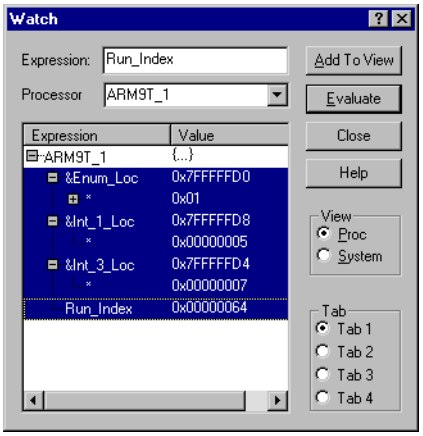
An alternative method of examining a variable is to use a **Watch** processor view. This allows you to see the memory address of the variable as well as its value. In this example you start by reloading and starting the current program, then stopping it:

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 first executable line of source code in function `main()`.

3. 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.

4. 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 enter a valid variable name preceded by an ampersand (&). See Figure 3-5.



Expression: Run_Index
Processor: ARM9T_1

Expression	Value
ARM9T_1	{...}
&Enum_Loc	0x7FFFFFD0
*	0x01
&Int_1_Loc	0x7FFFFFD8
*	0x00000005
&Int_3_Loc	0x7FFFFFD4
*	0x00000007
Run_Index	0x00000064

View: [✓ Proc] System
Tab: [✓ Tab 1] Tab 2 Tab 3 Tab 4

Figure 3-5 Specifying variables to watch

Enter the first expression by typing:

```
&Enum_Loc
```

then either 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.

Enter, in a similar way:

```
&Int_1_Loc
&Int_3_Loc
Run_Index
```

and expand their lines also.

The `Run_Index` variable name is not preceded by an ampersand because, in this program, the variable is stored in a hardware 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.

5. Select all the lines you have entered, as shown in Figure 3-5, ensure that Proc is the selected View and Tab1 the selected Tab, then click the **Add To View** button and the **Close** button.

6. 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 **Formats** → **Dec** so that the value of `Run_Index` is displayed as a decimal number.

7. Press **F5** again, to continue program execution, and enter a number not much greater than 750 this time for the number of runs required, say 770. The program performs the first 750 executions of the Dhrystone test and stops at the breakpoint you defined in *Setting a breakpoint* on page 3-4.

8. 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.

9. 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.

10. 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.

3.4 Examining the contents of registers

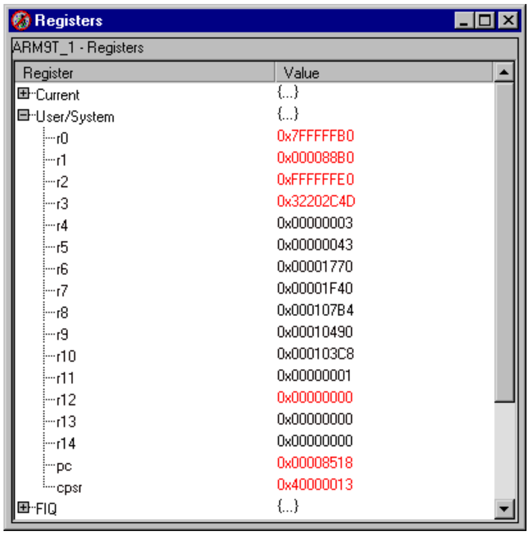
To examine the contents of registers used by the currently loaded program:

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 first executable line of source code in 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 3-6.



Register	Value
⊞ Current	{...}
⊟ User/System	{...}
r0	0x7FFFFFB0
r1	0x000088B0
r2	0xFFFFFFE0
r3	0x32202C4D
r4	0x00000003
r5	0x00000043
r6	0x00001770
r7	0x00001F40
r8	0x000107B4
r9	0x00010490
r10	0x000103C8
r11	0x00000001
r12	0x00000000
r13	0x00000000
r14	0x00000000
pc	0x00008518
cpsr	0x40000013
⊞ FIQ	{...}

Figure 3-6 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 color.

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 at which you are prompted, in the Console processor view, for the number of runs to perform. You need enter only a very small number 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.

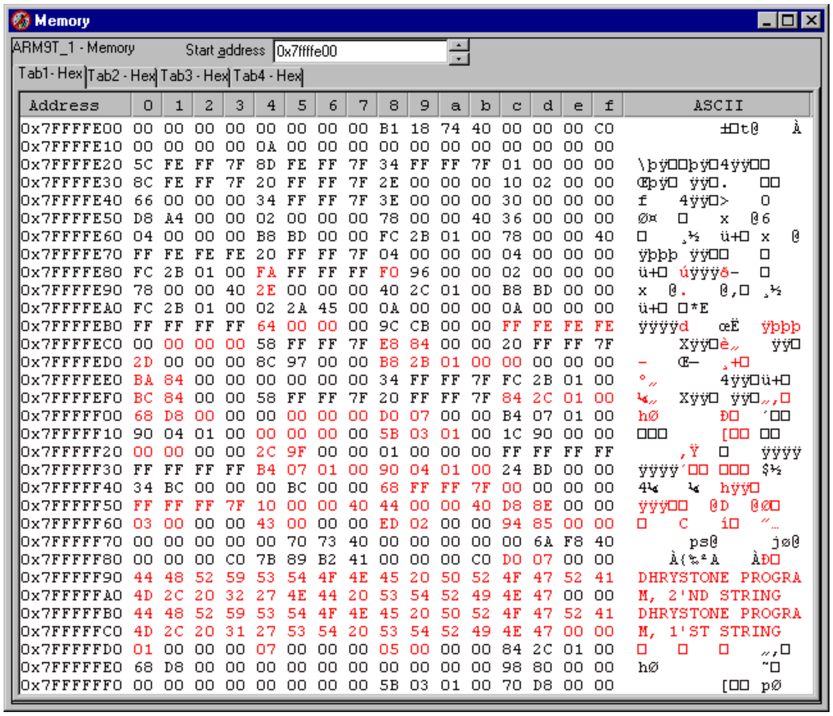
   If you select **Add to System** from the pop-up menu, the currently selected register is added to those that are displayed in the Registers system view. This is of particular value when your target has multiple processors and you want to examine the contents of some registers of each processor. Collecting 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.

3.5 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 first executable line of source code in function `main()`.
3. Select **Memory** from the **Processor Views** menu and reposition or resize the window if necessary. Figure 3-7 shows a typical memory processor view.



```
Address     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f       ASCII
0x7FFFFE00 00 00 00 00 00 00 00 00 B1 18 74 40 00 00 00 C0         ±□t@    À
0x7FFFFE10 00 00 00 00 0A 00 00 00 00 00 00 00 00 00 00 00
0x7FFFFE20 5C FE FF 7F 8D FE FF 7F 34 FF FF 7F 01 00 00 00  \þÿ□□þÿ□4ÿÿ□□
0x7FFFFE30 8C FE FF 7F 20 FF FF 7F 2E 00 00 00 10 02 00 00  Œþÿ□ ÿÿ□.   □□
0x7FFFFE40 66 00 00 00 34 FF FF 7F 3E 00 00 00 30 00 00 00  f   4ÿÿ□>   0
0x7FFFFE50 D8 A4 00 00 02 00 00 00 78 00 00 40 36 00 00 00  Ø¤  □   x  @6
0x7FFFFE60 04 00 00 00 B8 BD 00 00 FC 2B 01 00 78 00 00 40  □   ¸½  ü+□ x  @
0x7FFFFE70 FF FE FE FE 20 FF FF 7F 04 00 00 00 04 00 00 00  ÿþþþ ÿÿ□□   □
0x7FFFFE80 FC 2B 01 00 FA FF FF FF F0 96 00 00 02 00 00 00  ü+□ úÿÿÿð–  □
0x7FFFFE90 78 00 00 40 2E 00 00 00 40 2C 01 00 B8 BD 00 00  x  @.   @,□ ¸½
0x7FFFFEA0 FC 2B 01 00 02 2A 45 00 0A 00 00 00 0A 00 00 00  ü+□ □*E
0x7FFFFEB0 FF FF FF FF 64 00 00 00 9C CB 00 00 FF FE FE FE  ÿÿÿÿd   œË  ÿþþþ
0x7FFFFEC0 00 00 00 00 58 FF FF 7F E8 84 00 00 20 FF FF 7F      Xÿÿ□è„   ÿÿ□
0x7FFFFED0 2D 00 00 00 8C 97 00 00 B8 2B 01 00 00 00 00 00  -   Œ—  ¸+□
0x7FFFFEE0 BA 84 00 00 00 00 00 00 34 FF FF 7F FC 2B 01 00  º„      4ÿÿ□ü+□
0x7FFFFEF0 BC 84 00 00 58 FF FF 7F 20 FF FF 7F 84 2C 01 00  ¼„  Xÿÿ□ ÿÿ□„,□
0x7FFFFF00 68 D8 00 00 00 00 00 00 D0 07 00 00 B4 07 01 00  hØ      Ð□  ´□□
0x7FFFFF10 90 04 01 00 00 00 00 00 5B 03 01 00 1C 90 00 00  □□□     [□□ □□
0x7FFFFF20 00 00 00 00 2C 9F 00 00 01 00 00 00 FF FF FF FF      ,Ÿ  □   ÿÿÿÿ
0x7FFFFF30 FF FF FF FF B4 07 01 00 90 04 01 00 24 BD 00 00  ÿÿÿÿ´□□ □□□ $½
0x7FFFFF40 34 BC 00 00 00 BC 00 00 68 FF FF 7F 00 00 00 00  4¼   ¼  hÿÿ□
0x7FFFFF50 FF FF FF 7F 10 00 00 40 44 00 00 40 D8 8E 00 00  ÿÿÿ□□  @D  @Ø□
0x7FFFFF60 03 00 00 00 43 00 00 00 ED 02 00 00 94 85 00 00  □   C   í□  ”…
0x7FFFFF70 00 00 00 00 00 70 73 40 00 00 00 00 00 6A F8 40       ps@     jø@
0x7FFFFF80 00 00 00 C0 7B 89 B2 41 00 00 00 C0 D0 07 00 00     À{‰²A   ÀÐ□
0x7FFFFF90 44 48 52 59 53 54 4F 4E 45 20 50 52 4F 47 52 41  DHRYSTONE PROGRA
0x7FFFFFA0 4D 2C 20 32 27 4E 44 20 53 54 52 49 4E 47 00 00  M, 2'ND STRING
0x7FFFFFB0 44 48 52 59 53 54 4F 4E 45 20 50 52 4F 47 52 41  DHRYSTONE PROGRA
0x7FFFFFC0 4D 2C 20 31 27 53 54 20 53 54 52 49 4E 47 00 00  M, 1'ST STRING
0x7FFFFFD0 01 00 00 00 07 00 00 00 05 00 00 00 84 2C 01 00  □   □   □   „,□
0x7FFFFFE0 68 D8 00 00 00 00 00 00 00 00 00 00 98 80 00 00  hØ          ˜□
0x7FFFFFF0 00 00 00 00 00 00 00 00 5B 03 01 00 70 D8 00 00          [□□ pØ
```

Figure 3-7 Examining contents of memory

You saw in *Addresses and contents of variables* on page 3-7 that memory addresses of interest were in the region of `0x7ffffd0`, so set the Start address value to, say, `0x7ffffe00`.

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 Memory processor view are displayed in color.

5. 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.

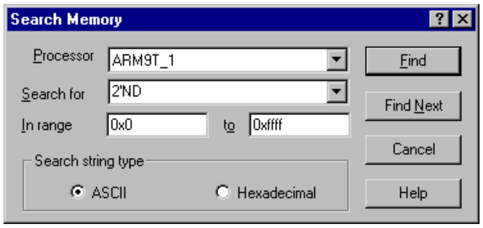
   You soon reach the point at which you are prompted, in the Console processor view, for the number of runs to perform. You need enter only a very small number this time.

6. Explore the format options available from the Memory processor view pop-up menu. The Size and Format settings appear both as items on the pop-up menu and as radio buttons in the dialog displayed when you select Properties... from the pop-up menu. More information about these options is given in Chapter 5 *AXD Desktop*.

3.6 Locating and changing values and verifying changes

To locate a value (of a variable or string, for example) in memory, change it, and continue execution in order to verify the effects of the change:

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 first executable line of source code in function `main()`.
3. Select **Memory** from the **Search** menu, enter `2'ND` in the **Search for** field, set the **In range** addresses to `0x0` and `0xffff`, select **ASCII** for the **Search string type**, and click the **Find** button, as shown in Figure 3-8.



Search Memory

Processor: ARM9T_1

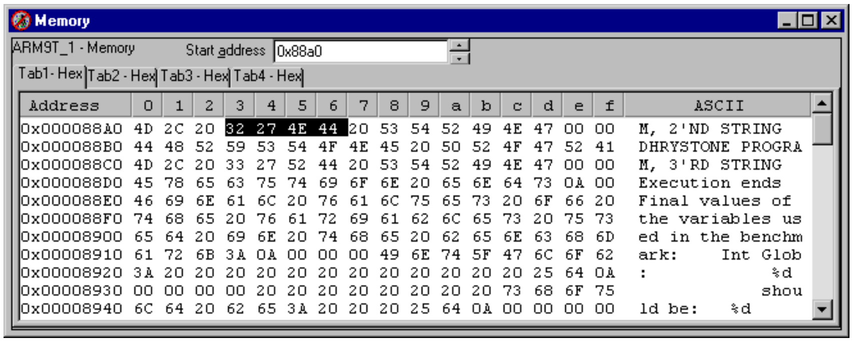
Search for: 2'ND

In range: 0x0 to 0xffff

Search string type: [✓ ASCII] Hexadecimal

Figure 3-8 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 3-9.



Memory

ARM9T_1 - Memory   Start address: 0x88a0

```
Address     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f   ASCII
0x000088A0 4D 2C 20 32 27 4E 44 20 53 54 52 49 4E 47 00 00   M, 2'ND STRING
0x000088B0 44 48 52 59 53 54 4F 4E 45 20 50 52 4F 47 52 41   DHRYSTONE PROGRA
0x000088C0 4D 2C 20 33 27 52 44 20 53 54 52 49 4E 47 00 00   M, 3'RD STRING
0x000088D0 45 78 65 63 75 74 69 6F 6E 20 65 6E 64 73 0A 00   Execution ends
0x000088E0 46 69 6E 61 6C 20 76 61 6C 75 65 73 20 6F 66 20   Final values of
0x000088F0 74 68 65 20 76 61 72 69 61 62 6C 65 73 20 75 73   the variables us
0x00008900 65 64 20 69 6E 20 74 68 65 20 62 65 6E 63 68 6D   ed in the benchm
0x00008910 61 72 6B 3A 0A 00 00 00 49 6E 74 5F 47 6C 6F 62   ark:    Int Glob
0x00008920 3A 20 20 20 20 20 20 20 20 20 20 20 20 25 64 0A   :             %d
0x00008930 00 00 00 00 20 20 20 20 20 20 20 20 73 68 6F 75               shou
0x00008940 6C 64 20 62 65 3A 20 20 20 25 64 0A 00 00 00 00   ld be:    %d
```

Figure 3-9 Changing contents of memory

You might need to right-click in the window to display the pop-up menu and set **Size** to `8 bit` and **Format** to `Hex`.

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

   Double-click on the value `32`, type `4E` and press Return.

Double-click on the value `27`, type `4F` and press Return.

Double-click on the value `4E`, type `2E` and press Return.

Double-click on the value `44`, type `32` 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.

3.7 Creating a revised version of the program

In the previous example, 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 and should be included permanently.

How to do that is beyond the scope of this book. It usually involves changes to the source code of your program, followed by recompiling and relinking. It could 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 CodeWarior 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:

- stopping execution at a breakpoint
- stopping execution at a watchpoint
- stepping through a program.

Detailed descriptions of how to use these facilities are given in *Execute menu* on page 5-49, 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
- 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 *Watch/Breakpoints...* on page 5-50, and toggling (switching on and off) in *Toggle Breakpoint* on page 5-51.

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 *Watch/Breakpoints...* on page 5-50.

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 5-49.

4.2 Expressions

This section describes:

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

4.2.1 Using expressions

You use expressions when you define watches in a **Watch** processor view or a **Watch** system view. Such an expression might be simply the name of a variable, but could, 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, restrictions on the use of certain C++ constructions in AXD expressions.

Expression guidelines

The following rules apply to expression evaluation in AXD:

- Member functions of C++ classes cannot be used in expressions.
- Overloaded functions cannot be used in expressions.
- Only C operators can be used in constructing expressions. Any operators defined in a C++ class that also have a meaning in C (such as `[]`) will not work correctly because AXD uses the C operator instead. Specific C++ operators, such as the scope operator `::`, are not recognized.

- Base classes cannot be accessed 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 will give undefined results.

- **`private`**, **`public`**, and **`protected`** attributes are not recognized in AXD expression evaluation. This means that private and protected member variables can be used during expression evaluation because AXD treats them as public.

4.2.3 Expression examples

Examples of expressions that would be 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*
- *Source files*
- *Disassembled code* on page 4-7
- *Registers* on page 4-7
- *Watch* on page 4-8
- *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 4-9
- *Debugger internals* on page 4-10
- *Backtrace* on page 4-10
- *Communications channel* on page 4-10
- *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.

You can view, and possibly edit, the following types of data:

4.3.1 Control

The main **Control** 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 5-33.

4.3.2 Source files

To display the source code that gave rise to 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 in order 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 **Open File**.
5. Right-click in the resulting view of the source file to display another pop-up menu which includes the ability to interleave disassembled code in the listing of the source file.

For further details see *Source... processor view* on page 5-29.

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 5-27.

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 5-14. To select registers from various **Registers** processor views to display together in a single **Registers** system view, see *Registers system view* on page 5-39.

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 the values of 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 on which you can 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, and select **Add Watch**. In the resulting **Watch** dialog, shown in both *Watch processor view* on page 5-16 and *Watch system view* on page 5-40, 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.

Although you can specify expressions to be watched, 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 5-18.

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 5-21.

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 5-42.

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, 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 can refer to the same address. Any code between the addresses referred to by the low-level and high-level symbols generally concerns the stack backtrace structure in procedures that conform to the appropriate variants of the *ARM/Thumb Procedure Call Standard* (ATPCS), or argument lists in other procedures. For information on ATPCS, see *ADS Tools Guide*. To display a list of the low-level symbols in your program, use the **Low Level Symbols** processor view.

In a regular expression, indicate high-level and low-level symbols as follows:

- precede the symbol with `@` to indicate a low-level symbol
- precede the symbol with `^` to indicate a high-level symbol.

For further information, see *Low Level Symbols processor view* on page 5-23.

4.3.10 Debugger internals

Various internal variables contain information relevant to the current debugging session. Also, when you target the ARMulator, 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 an emulated target only).

For further information, see *Debugger Internals system view* on page 5-45.

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 5-19.

4.3.12 Communications channel

The **Comms Channel** processor view provides you with the facility 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 is also a facility 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 5-25.

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, and to display on your screen output that might normally be sent elsewhere.

For further information, see *Console processor view* on page 5-26.

4.4 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 5-5) or before reloading the image (see Figure 5-55 on page 5-37).

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 Tools Guide*, to analyze it.

———————————

To collect information on just a 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. In Chapter 2 *Getting Started in AXD* you learnt how to use 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. Other sections of this chapter describe each main menu in more detail.

The first screen you see when you start running AXD is similar to the screen shown in Figure 5-1.

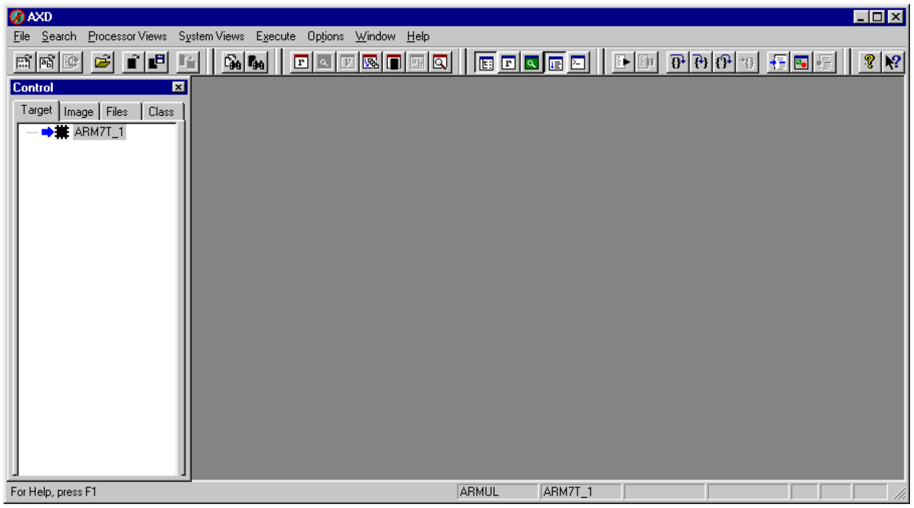


Figure 5-1 AXD opening screen

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 5-13 and *System Views menu* on page 5-33 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 5-55). 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.

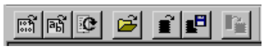


Figure 5-2 File toolbar

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

Search toolbar

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

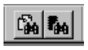


Figure 5-3 Search toolbar

These tools are described as menu items in *Search menu* on page 5-11.

Processor Views toolbar

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

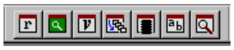


Figure 5-4 Processor Views toolbar

These tools are described as menu items in *Processor Views menu* on page 5-13.

System Views toolbar

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

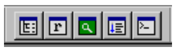


Figure 5-5 System Views toolbar

These tools are described as menu items in *System Views menu* on page 5-33.

Execute toolbar

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

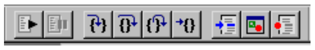


Figure 5-6 Execute toolbar

These tools are described as menu items in *Execute menu* on page 5-51.

Help toolbar

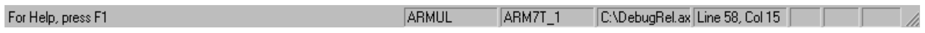
Help toolbar icons provide two ways of accessing AXD online help items, as shown in Figure 5-7.

Figure 5-7 Help toolbar

These tools are described in *Help menu* on page 5-67.

5.1.3 Status bar

If you choose to display the status bar (see page 5-63) it appears at the bottom of the AXD screen, as shown in Figure 5-8.



For Help, press F1 | ARMUL | ARM7T_1 | C:\DebugRel.ax | Line 58, Col 15

Figure 5-8 Status bar

Help text is displayed at the left end 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.

The remainder of the status bar shows the current debug agent, current processor, and current image. Also, when a source or disassembly view has the focus, it shows the current cursor position in line and column format.

5.2 File menu

File menu items, described in the following subsections, 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.

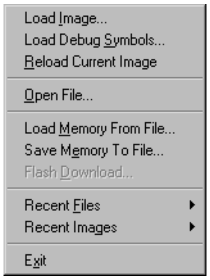
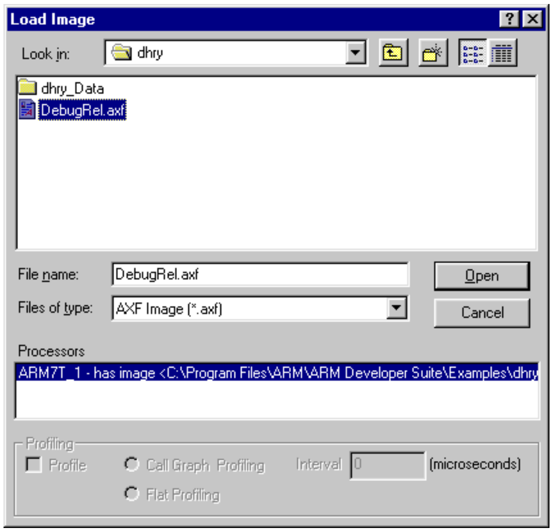


Figure 5-9 File menu

5.2.1 Load Image...

To select a file containing an image that you want 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)
ARM7T_1 - has image <C:\Program Files\ARM\ARM Developer Suite\Examples\dhry
Interval 0 (microseconds)

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 should be 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 make any other profiling settings you want in this dialog before you load the image:

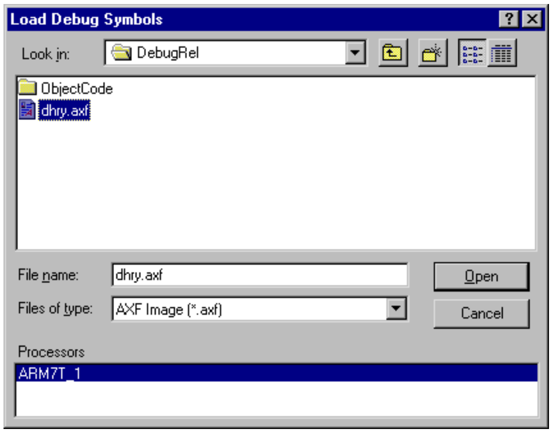
- Flat profiling, with a large number of microseconds between samples, imposes the least overhead on execution time, but might not give accurate enough results.
- Call-graph profiling, at short intervals, gives the most comprehensive and accurate information, but at greater cost in execution time.

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 4-12).

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

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.


Load Debug Symbols

Look in: DebugRel

ObjectCode
dhry.axf

File name: dhry.axf
Files of type: AXF Image (*.axf)

Processors
ARM7T_1

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.

5.2.3 Reload Current Image

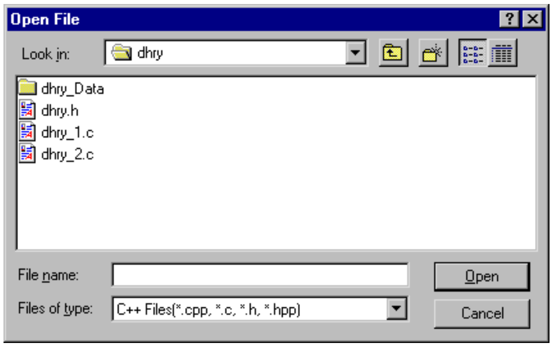
Having finished executing an image, the simplest way of preparing it for re-execution is to reload it.

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 5-55 on page 5-38).

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.


Open File

Look in: dhry

dhry_Data
dhry.h
dhry_1.c
dhry_2.c

File name:

Files of type: C++ Files(*.cpp, *.c, *.h, *.hpp)

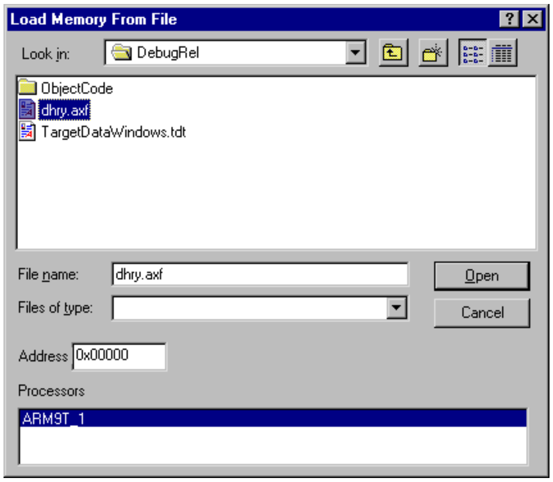
Figure 5-12 Selecting a source file to open

Navigate to the directory in which the file is stored. You can specify that only files with a particular filename extension should be 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

Address 0x00000

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 on page 5-9. 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.

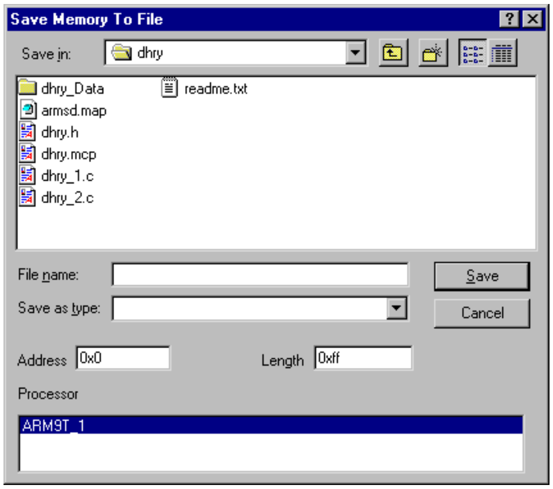


Figure 5-14 Saving memory contents in a file

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.

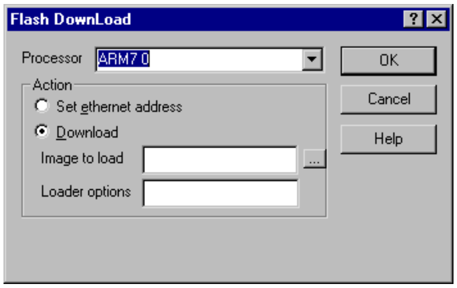


Figure 5-15 Flash Download dialog

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 need to do this if you have built your own Angel port with a fixed Ethernet address.

Refer to the *ADS Tools Guide* for more information on Flash downloading.

5.2.8 Recent Files

If you have opened any files by selecting **Open File...** from the **File** menu and using the resulting browse dialog, you can open any of those files again more easily by pointing to **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.

5.2.9 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, point to **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.

5.2.10 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 5-16, allows you to search for specific contents, either in a source file related to a current process or in memory.

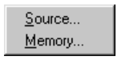
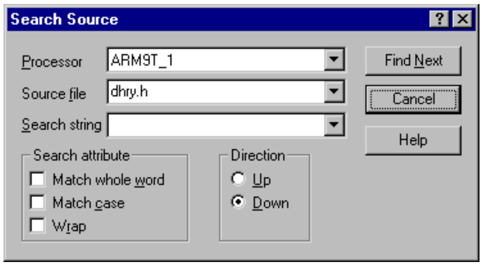


Figure 5-16 Search menu

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 5-17, allows you to specify the target character string, and the file to be searched.



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

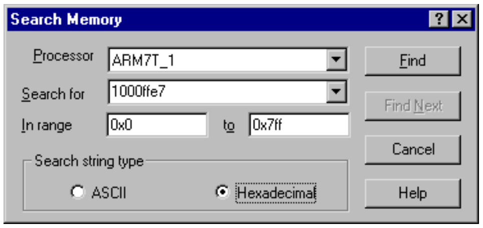
Figure 5-17 Searching for a string in a source file

You can search upwards or downwards, and specify case sensitivity, and whether whole words only and wrapped strings should be considered.

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 5-18, allows you to specify what to search for and where to search.



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

Figure 5-18 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. Be sure to 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 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 5-19, allows you to examine and change information relating to specific processors.

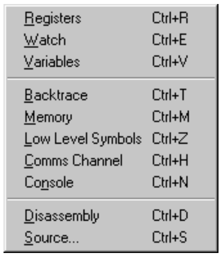


Figure 5-19 Processor Views menu

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 5-33). 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, you should consider using a corresponding system view instead (see *System Views menu* on page 5-33).

Descriptions follow of all the **Processor Views** menu items.

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.

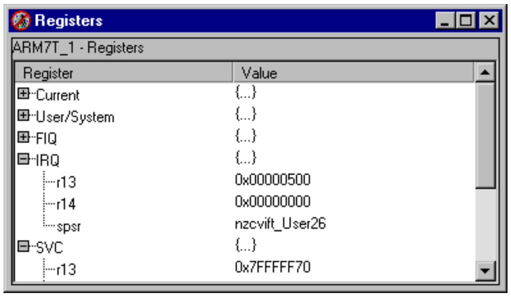
The description that follows of each processor view includes a reproduction of its pop-up menu. Online help gives further details.

5.4.1 Registers processor view

The **Registers** processor view allows you to examine and change the value of any of the registers in a specific processor.

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 5-20.

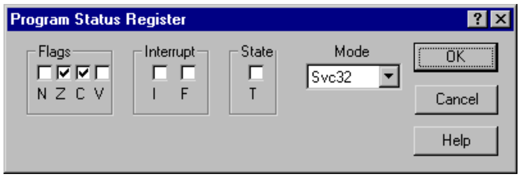


```
Register          Value
⊞ Current         {...}
⊞ User/System     {...}
⊞ FIQ             {...}
⊟ IRQ             {...}
    r13           0x00000500
    r14           0x00000000
    spsr          nzcvift_User26
⊟ SVC             {...}
    r13           0x7FFFFF70
```

Figure 5-20 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.

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, causes the display of the dialog shown in Figure 5-21.



Flags: N ☐ Z ✓ C ✓ V ☐
Interrupt: I ☐ F ☐
State: T ☐
Mode: Svc32

Figure 5-21 Program Status Register dialog

The ARM9E processor has an extra bit in its PSR, not visible in the Registers processor view shown in Figure 5-20 or in the PSR dialog shown in Figure 5-21.

If you are using an ARM9E processor, refer to *Registers processor view pop-up menu* to find how to change the format for displaying the PSR to *Enhanced PSR* (E-PSR). The extra bit (Q, signifying saturation) is now visible in the Registers processor view and, if you edit the value of that register, the dialog in Figure 5-22 is displayed.

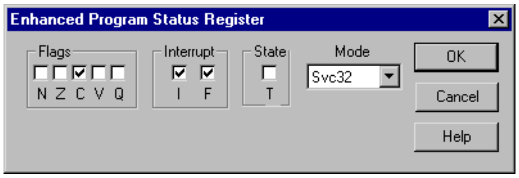


Figure 5-22 Enhanced Program Status Register dialog

Registers processor view pop-up menu

To display the **Registers** pop-up menu, shown in Figure 5-23, right-click within the **Registers** processor view.

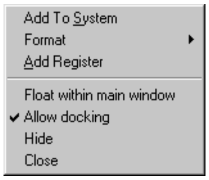


Figure 5-23 Registers processor view pop-up menu

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 5-24.

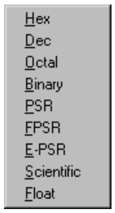


Figure 5-24 Formats available for displaying registers

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

To add one of the registers displayed in a **Registers** processor view to the **Registers** system view (see *Registers system view* on page 5-40), right-click on the required register to select it and display the pop-up menu, then select **Add to System**.

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

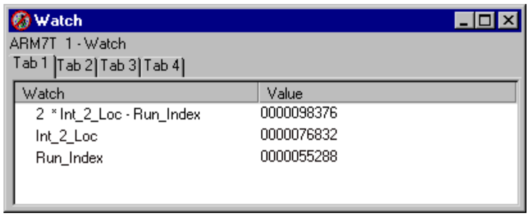
The **Watch** processor view allows you to examine the value of variables, or of expressions dependent on variables, in an image being executed by a specific processor.

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. The menu item is disabled if the processor has no associated image.

A **Watch** processor view is initially empty. You choose what is to be listed and have its value shown. To add a line to this view, select **Add Watch** from the pop-up menu (see Figure 5-26 on page 5-17). 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.

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, as well as a number of variables and constants.

A typical **Watch** processor view is shown in Figure 5-25.



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

Figure 5-25 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 whichever page you want to view.

Watch processor view pop-up menu

To display the **Watch** pop-up menu, shown in Figure 5-26, right-click within the **Watch** processor view.

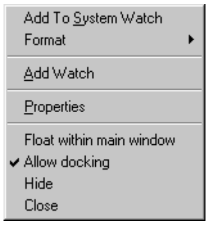


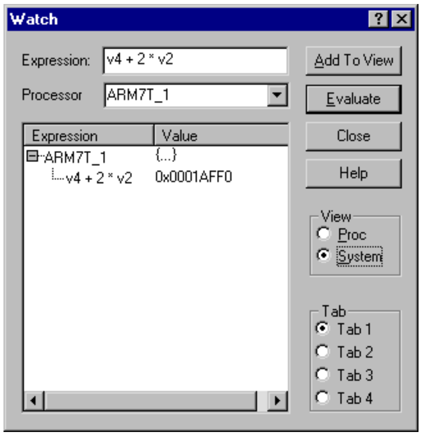
Figure 5-26 Watch processor view pop-up menu

Refer to AXD online help for full details.

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.

To define a new watch, select **Add Watch** from the pop-up menu. The resulting dialog is shown in Figure 5-27.



Expression: v4 + 2 * v2
Processor: ARM7T_1

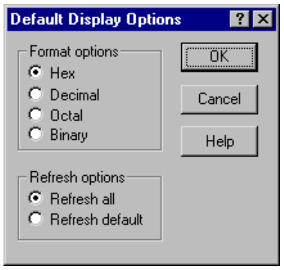
Expression	Value
ARM7T_1	{...}
v4 + 2 * v2	0x0001AFF0

View: Proc [✓ System]
Tab: [✓ Tab 1] Tab 2 Tab 3 Tab 4

Figure 5-27 Watch dialog

Enter a new expression to watch. Specify the processor, whether the new watch should be added to the **Watch** processor view or system view (see *Watch system view* on page 5-41), and on which tabbed page it should appear.

To display the dialog shown in Figure 5-28, select **Properties...** from the pop-up menu:


Default Display Options

Format options
- [✓ Hex]
- Decimal
- Octal
- Binary

Refresh options
- [✓ Refresh all]
- Refresh default

Figure 5-28 Default Display Options dialog

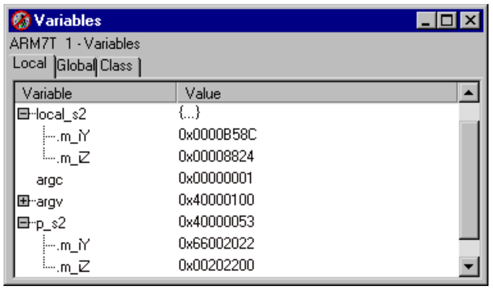
5.4.3 Variables processor view

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, being those with scope within the current function
- **Global** variables, being those with scope over all parts of the program
- **Class** variables, being those with scope within the current class only.

A typical **Variables** processor view is shown in Figure 5-29, with its Global tab selected.


Variables

ARM7T 1 - Variables

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

Figure 5-29 Variables processor view

Click on the + or – boxes to expand or collapse each level of the displayed tree structure.

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.

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.

Variables processor view pop-up menu

To display the Variables pop-up menu, shown in Figure 5-30, right-click within the **Variables** processor view.

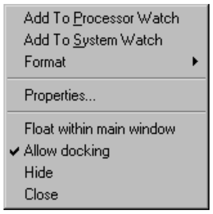


Figure 5-30 Variables processor view pop-up menu

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.

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

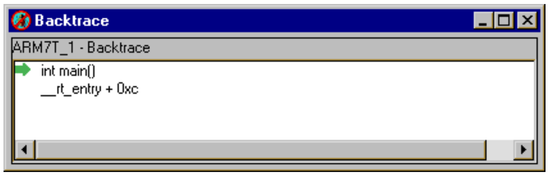
To display the dialog shown in Figure 5-28, select **Properties...** from the pop-up menu.

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 5-31.



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

Figure 5-31 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.

Backtrace processor view pop-up menu

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

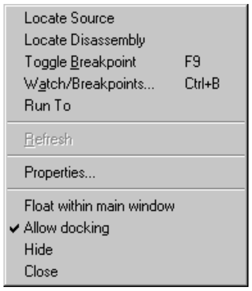
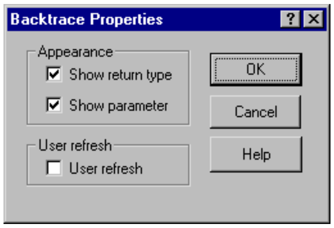


Figure 5-32 Backtrace processor view pop-up menu

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 (except **Watch/Breakpoints...** which remains enabled).

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

To display the dialog shown in Figure 5-33, select **Properties...** from the pop-up menu.



- [✓] Show return type
- [✓] Show parameter
- [ ] User refresh

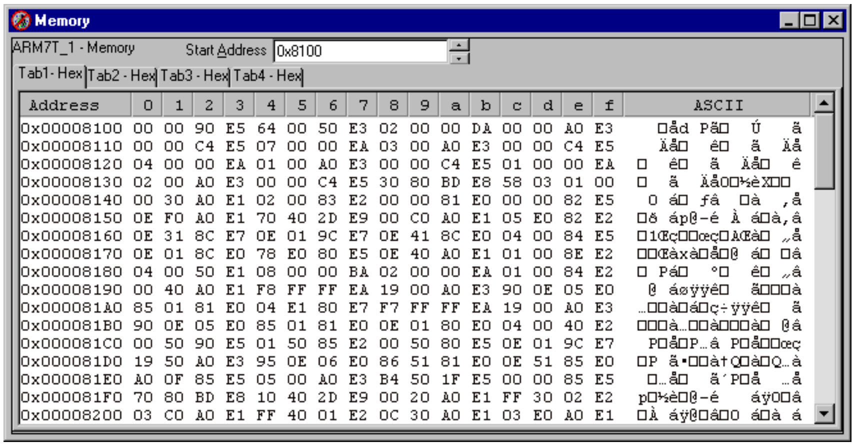
Figure 5-33 Backtrace Properties dialog

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 5-34.



Start Address 0x8100

```
Address     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f        ASCII
0x00008100 00 00 90 E5 64 00 50 E3 02 00 00 DA 00 00 A0 E3    □åd Pã□  Ú   ã
0x00008110 00 00 C4 E5 07 00 00 EA 03 00 A0 E3 00 00 C4 E5    Äå□  ê□  ã  Äå
0x00008120 04 00 00 EA 01 00 A0 E3 00 00 C4 E5 01 00 00 EA  □  ê□  ã  Äå□  ê
0x00008130 02 00 A0 E3 00 00 C4 E5 30 80 BD E8 58 03 01 00  □  ã  Äå0□½èX□□
0x00008140 00 30 A0 E1 02 00 83 E2 00 00 81 E0 00 00 82 E5   0 á□ fâ  □à  ,å
0x00008150 0E F0 A0 E1 70 40 2D E9 00 C0 A0 E1 05 E0 82 E2  □ð áp@-é À á□à,â
0x00008160 0E 31 8C E7 0E 01 9C E7 0E 41 8C E0 04 00 84 E5  □1Œç□□œç□AŒà□ „å
0x00008170 0E 01 8C E0 78 E0 80 E5 0E 40 A0 E1 01 00 8E E2  □□Œàxà□å□@ á□ □â
0x00008180 04 00 50 E1 08 00 00 BA 02 00 00 EA 01 00 84 E2  □ Pá□  °□  ê□ „â
0x00008190 00 40 A0 E1 F8 FF FF EA 19 00 A0 E3 90 0E 05 E0   @ áøÿÿê□  ã□□□à
0x000081A0 85 01 81 E0 04 E1 80 E7 F7 FF FF EA 19 00 A0 E3  …□□à□á□ç÷ÿÿê□  ã
0x000081B0 90 0E 05 E0 85 01 81 E0 0E 01 80 E0 04 00 40 E2  □□□à…□□à□□□à□ @â
0x000081C0 00 50 90 E5 01 50 85 E2 00 50 80 E5 0E 01 9C E7   P□å□P…â P□å□□œç
0x000081D0 19 50 A0 E3 95 0E 06 E0 86 51 81 E0 0E 51 85 E0  □P ã•□□à†Q□à□Q…à
0x000081E0 A0 0F 85 E5 05 00 A0 E3 B4 50 1F E5 00 00 85 E5   □…å□  ã´P□å  …å
0x000081F0 70 80 BD E8 10 40 2D E9 00 20 A0 E1 FF 30 02 E2  p□½è□@-é  áÿ0□â
0x00008200 03 C0 A0 E1 FF 40 01 E2 0C 30 A0 E1 03 E0 A0 E1  □À áÿ@□â□0 á□à á
```

Figure 5-34 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.

Alternatively, again using **Properties...** from the Memory pop-up menu, you can display the memory contents as disassembled code in ARM, Thumb, or mixed format. In these formats each line of the display shows the contents of a 32-bit or 16-bit word.

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.

You can open multiple memory views, even on a single processor, if you need more than four tabbed pages.

Memory processor view pop-up menu

To display the Memory pop-up menu, shown in Figure 5-35, right-click within the **Memory** processor view.

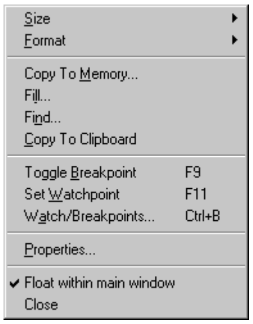


Figure 5-35 Memory processor view pop-up menu

Toggle Breakpoint

This toggles 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.

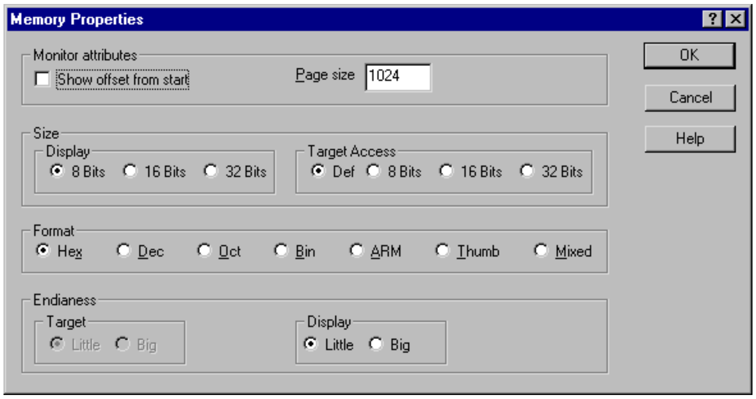
Set Watchpoint

This sets a watchpoint at the address defined by the current cursor position. If a watchpoint already exists at this address it is replaced. If no watchpoint exists at this address a default watchpoint is created here.

Watch/Breakpoints...

This displays a dialog allowing you to create, edit, or delete a watchpoint or breakpoint.

Refer to AXD online help for details of the other Memory pop-up menu items, including the Memory Properties dialog, shown in Figure 5-36.


Memory Properties

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

Format: [✓ Hex] Dec Oct Bin ARM Thumb Mixed

Endianess — Target: [✓ Little] Big — Display: [✓ Little] Big

Figure 5-36 Memory Properties dialog

Data width for memory reads and writes

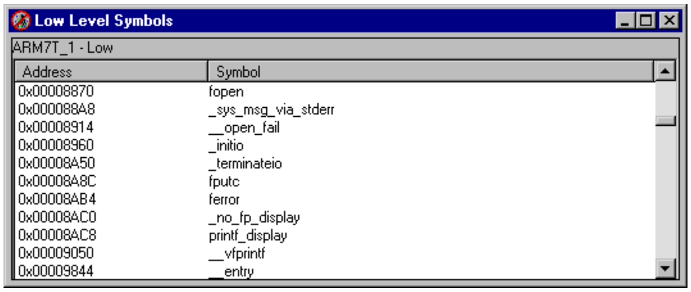
The **Target Access** group of radio buttons in the Memory Properties dialog allows you to specify the width of data read from or written to memory. Unless you have a particular requirement, use the **Def** setting to indicate that you want the debug engine to decide.

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 5-37.



```
ARM7T_1 - Low
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 5-37 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 5-38, right-click within the **Low Level Symbols** processor view.

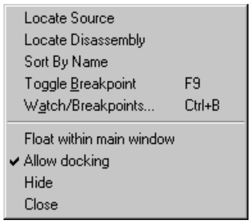


Figure 5-38 Low Level Symbols processor view pop-up menu

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.

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 communication channels, and to send data of your own. A channel viewer is supplied (`ThumbCV.dll`).

To select a channel viewer from those available, see *Multi-ICE configuration* on page 5-58 or *Remote_A configuration* on page 5-61.

The **Comms Channel** processor view enables you to use your selected channel viewer. A typical **Comms Channel Viewer** dialog is shown in Figure 5-39.

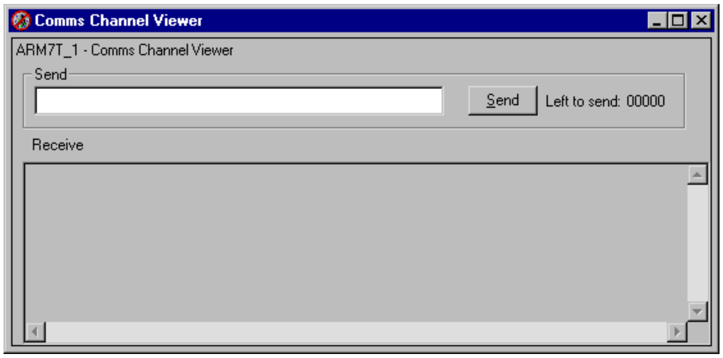


Figure 5-39 Comms Channel Viewer dialog

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.

Information received by the channel viewer is converted into ASCII character codes and displayed in the **Receive** window, if the channel viewer is active. However, if `0xffffffff` is received, the following word is treated and displayed as a number.

Comms Channel Viewer pop-up menu

To display the Comms Channel pop-up menu, shown in Figure 5-40, right-click within the **Comms Channel** processor view.

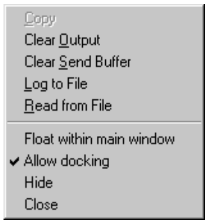


Figure 5-40 Comms Channel Viewer pop-up menu

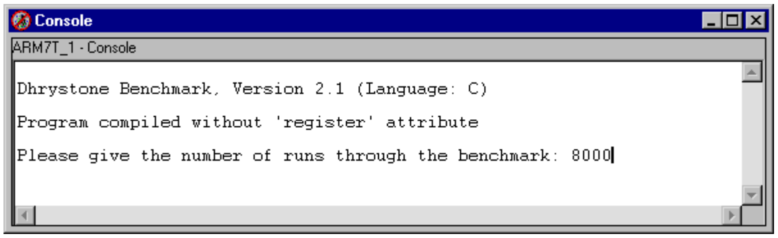
Refer to AXD online help for more details of these menu items.

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 5-41.



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

Figure 5-41 Console processor view

Console processor view pop-up menu

To display the Console pop-up menu, shown in Figure 5-42, right-click within the **Console** processor view.

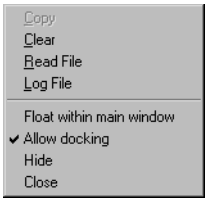


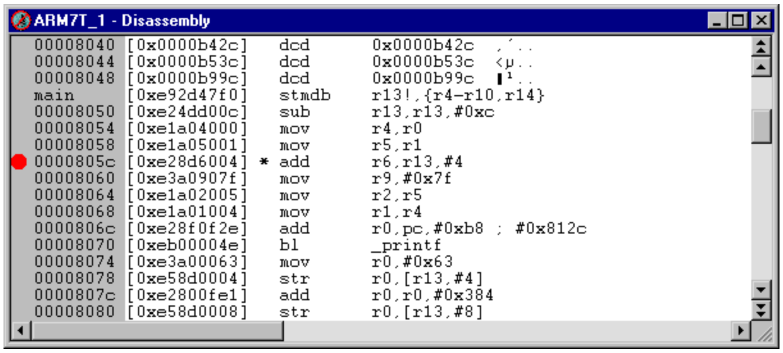
Figure 5-42 Console processor view pop-up menu

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 5-43. 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 5-45 on page 5-29).


ARM7T_1 - Disassembly

```
  00008040 [0x0000b42c]   dcd     0x0000b42c  ,´..
  00008044 [0x0000b53c]   dcd     0x0000b53c  <µ..
  00008048 [0x0000b99c]   dcd     0x0000b99c  ▌¹..
  main     [0xe92d47f0]   stmdb   r13!,{r4-r10,r14}
  00008050 [0xe24dd00c]   sub     r13,r13,#0xc
  00008054 [0xe1a04000]   mov     r4,r0
  00008058 [0xe1a05001]   mov     r5,r1
● 0000805c [0xe28d6004] * add     r6,r13,#4
  00008060 [0xe3a0907f]   mov     r9,#0x7f
  00008064 [0xe1a02005]   mov     r2,r5
  00008068 [0xe1a01004]   mov     r1,r4
  0000806c [0xe28f0f2e]   add     r0,pc,#0xb8 ; #0x812c
  00008070 [0xeb00004e]   bl      _printf
  00008074 [0xe3a00063]   mov     r0,#0x63
  00008078 [0xe58d0004]   str     r0,[r13,#4]
  0000807c [0xe2800fe1]   add     r0,r0,#0x384
  00008080 [0xe58d0008]   str     r0,[r13,#8]
```

Figure 5-43 Disassembly processor view

Disassembly processor view pop-up menu

To display the Disassembly pop-up menu, shown in Figure 5-44, right-click within the **Disassembly** processor view.

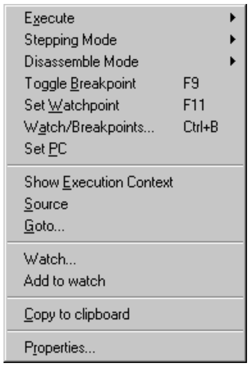


Figure 5-44 Disassembly processor view pop-up menu

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 5-51 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 **Disassemble Mode** on the pop-up menu.

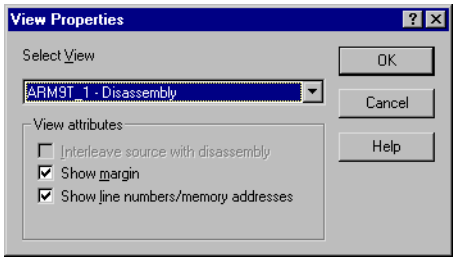
To activate or deactivate a breakpoint at the current cursor position, select **Toggle Breakpoint** from the pop-up menu. To set or replace a watchpoint on a currently selected value, select **Set Watchpoint** from the pop-up menu.

To create, edit, or delete a watchpoint or breakpoint, select **Watch/Breakpoints...** from the pop-up menu.

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

To display the dialog shown in Figure 5-45, select **Properties...** from the pop-up menu.

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


View Properties

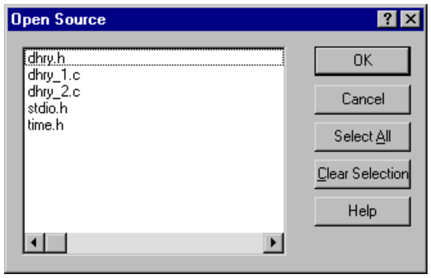
Select View: ARM9T_1 - Disassembly

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

Figure 5-45 Disassembly Properties dialog

5.4.10 Source... processor view

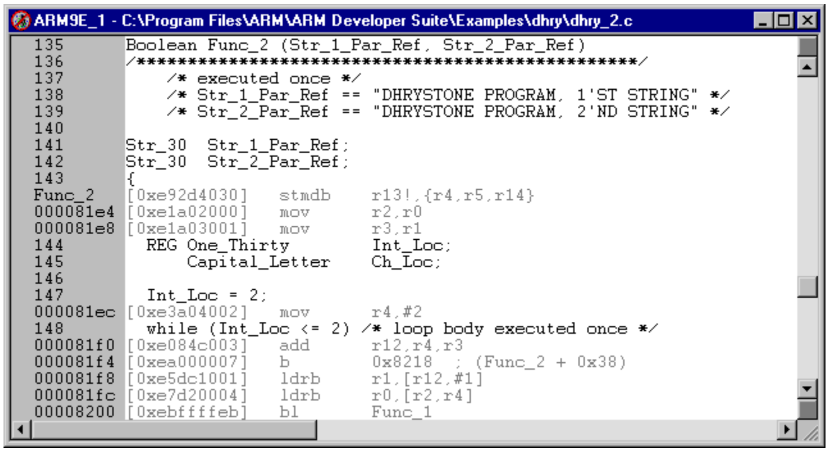
The **Source...** processor view first displays a file selection dialog, similar to that shown in Figure 5-46.


Open Source

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

Figure 5-46 Source file selection

This lists all the source files that have contributed debug information to the current image and allows you to select one to examine. (The list does not necessarily include all the source files used to build the image.) Select a filename and click the **OK** button to display the file, as shown in Figure 5-47.



```
ARM9E_1 - C:\Program Files\ARM\ARM Developer Suite\Examples\dhry\dhry_2.c
135         Boolean Func_2 (Str_1_Par_Ref, Str_2_Par_Ref)
136         /***************************************************/
137             /* executed once */
138             /* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1'ST STRING" */
139             /* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */
140
141         Str_30  Str_1_Par_Ref;
142         Str_30  Str_2_Par_Ref;
143         {
Func_2      [0xe92d4030]   stmdb    r13!,{r4,r5,r14}
000081e4    [0xe1a02000]   mov      r2,r0
000081e8    [0xe1a03001]   mov      r3,r1
144           REG One_Thirty        Int_Loc;
145               Capital_Letter    Ch_Loc;
146
147           Int_Loc = 2;
000081ec    [0xe3a04002]   mov      r4,#2
148           while (Int_Loc <= 2) /* loop body executed once */
000081f0    [0xe084c003]   add      r12,r4,r3
000081f4    [0xea000007]   b        0x8218  ; (Func_2 + 0x38)
000081f8    [0xe5dc1001]   ldrb     r1,[r12,#1]
000081fc    [0xe7d20004]   ldrb     r0,[r2,r4]
00008200    [0xebfffffeb]   bl       Func_1
```

Figure 5-47 Source... processor view

Figure 5-47 shows the kind of source file listing you see if you select **Interleave disassembly** from the pop-up menu.

Source... processor view pop-up menu

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

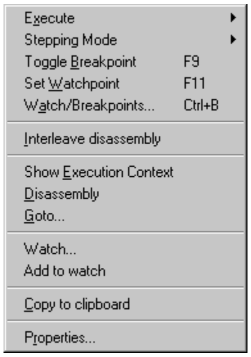


Figure 5-48 Source... processor view pop-up menu

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 5-51 for details of all but one of these items.

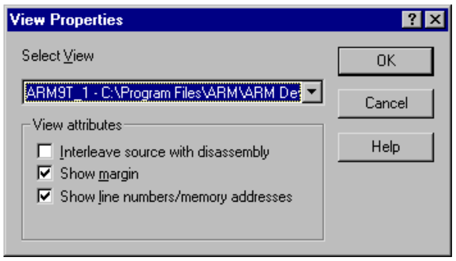
Set Next Statement is the item that appears on the **Execute** submenu and not in the **Execute** main menu. To resume execution at a specific statement, without executing any intervening statements, right-click on the required statement in the **Source...** processor view, point to **Execute** in the 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** Always step in disassembly instructions.
- **Strong source** Always step in source code statements.
- **Weak source** 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 5-49, select **Properties...** from the pop-up menu.

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



Select View

ARM9T_1 - C:\Program Files\ARM\ARM De

View attributes

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

Figure 5-49 Source View properties dialog

5.5 System Views menu

System views are not specific to any processor. Some system views show information about the whole system. Others help you reduce the number of views you need to display at one time.

A **Registers** system view, for example, can show registers that are associated with several processors. This means you can examine in a single system view registers that would 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 closed if it is currently open. System views that are open are checked on the menu. Figure 5-50 shows an example of a **System Views** menu.

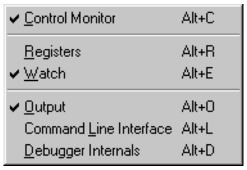


Figure 5-50 System Views menu

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 **System Views** menu items are described in the following subsections, which include reproductions of the pop-up menus. AXD online help gives further details.

5.5.1 Control system view

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

- **Target**
- **Image**
- **Files**
- **Class**.

Figure 5-51 shows a **Control** system view with its **Files** tab selected.

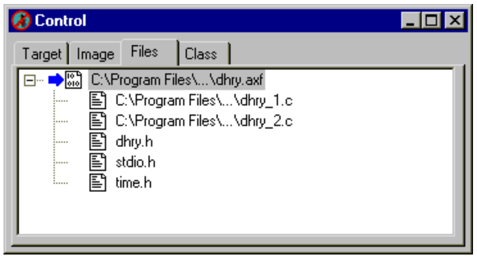

Control
Target
Image
Files
Class
C:\Program Files\...\dhry.axf
C:\Program Files\...\dhry_1.c
C:\Program Files\...\dhry_2.c
dhry.h
stdio.h
time.h

Figure 5-51 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 an 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 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.
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 5-52.

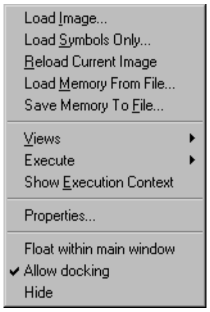
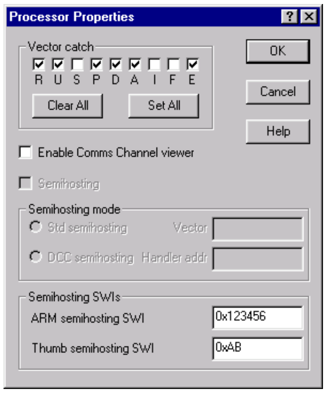


Figure 5-52 Pop-up menu when a processor is selected

Select **Properties...** from this pop-up menu to display the dialog shown in Figure 5-53.



Vector catch

R ✓ U ✓ S ☐ P ✓ D ✓ A ✓ I ☐ F ☐ E ✓

☐ 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 5-53 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 settings are stored in the debugger internal variable `$vector_catch` which has the default value of `%RUsPDAifE`. 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
`A`	Address (applied to 26-bit mode, so now never occurs)
`I`	*normal interrupt request* (IRQ)
`F`	*fast interrupt request* (FIQ)
`E`	Reserved (do not use).

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.

The **Comms Channel** and **Semihosting Mode** 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.

When you select a semihosting mode you must specify a vector. This is the position where the address is stored to which the program jumps following a semihosting *Software Interrupt* (SWI).

Under Semihosting SWIs, you specify an integer number identifying which ARM SWI and/or Thumb SWI is used for semihosting.

Image pop-up menu

With an image selected, the pop-up menu is as shown in Figure 5-54.

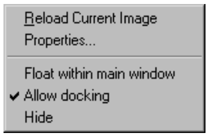


Figure 5-54 Pop-up menu when an image is selected

If you select **Properties...** from this pop-up menu, the dialog shown in Figure 5-55 is displayed.

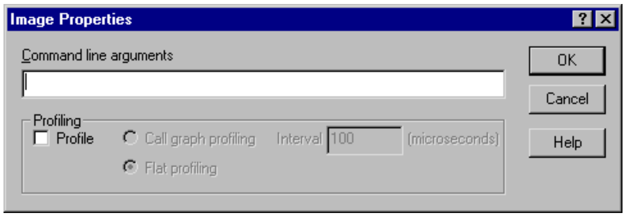


Figure 5-55 Image Properties dialog

The **Image Properties** dialog enables you to specify **Command-line arguments**. These are the arguments you would supply if you started 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 will become effective the next time you load or reload an image. You can change these settings to be as you want them when the next image execution begins. The settings shown are not necessarily those currently in force, because you may 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 5-56.

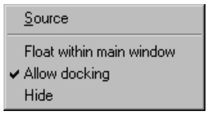


Figure 5-56 Pop-up menu when a file is selected

If you select **Source** from this pop-up menu, a Source processor view opens 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 5-57.

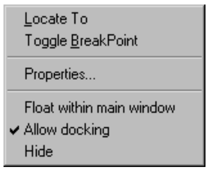
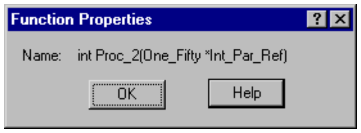


Figure 5-57 Pop-up menu when a function is selected

If you select **Properties...** from this pop-up menu, the dialog shown in Figure 5-58 is displayed.



Name: int Proc_2(One_Fifty *Int_Par_Ref)

Figure 5-58 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 5-59.

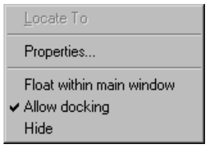
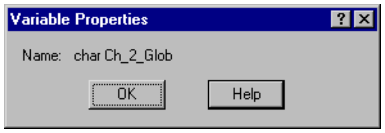


Figure 5-59 Pop-up menu when a variable is selected

If you select **Properties...** from this pop-up menu, the dialog shown in Figure 5-60 is displayed.



Name: char Ch_2_Glob

Figure 5-60 Variable Properties dialog

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

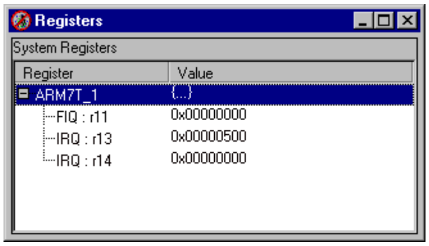
5.5.2 Registers system view

The **Registers** system view can display registers from more than one processor, and allows you to change their values.

If you are interested in seeing 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 the need to display 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.

Figure 5-61 shows a typical **Registers** system view:



Registers

System Registers

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

Figure 5-61 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 5-62, right-click within the **Registers** system view.

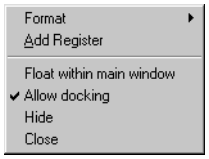


Figure 5-62 Registers system view pop-up menu

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 AXD online help for 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.

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.

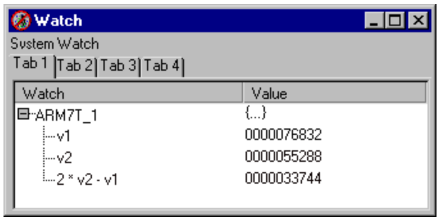
5.5.3 Watch system view

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.

A **Watch** system view is initially empty. You specify expressions. These expressions are evaluated each time program execution stops, and the values displayed. To add a line to this view, select **Add Watch** from the pop-up menu (see *Watch dialog* on page 5-42).

An expression can be simply the name of a variable. Expressions can also include logical and arithmetic operators as well as variable names and constants.

A typical **Watch** system view is shown in Figure 5-63.



Watch	Value
ARM7T_1	{...}
v1	0000076832
v2	0000055288
2 * v2 - v1	0000033744

Figure 5-63 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, it reappears in the state it was in when you hid it.

If you close a **Watch** system view then select it, it 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 5-64, right-click within the **Watch** system view.

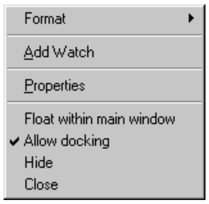
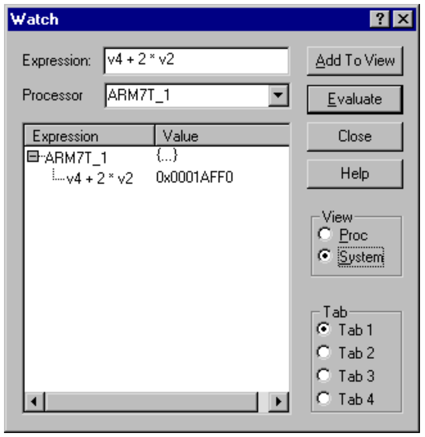


Figure 5-64 Watch system view pop-up menu

Refer to AXD online help for full details.

To display the dialog shown in Figure 5-65, select **Add Watch** from the pop-up menu.



Expression: v4 + 2 * v2
Processor: ARM7T_1

Expression	Value
ARM7T_1	{...}
v4 + 2 * v2	0x0001AFF0

View: System
Tab: Tab 1

Figure 5-65 Watch dialog

Enter a new expression to watch. Specify the processor, whether the new watch should be added to the **Watch** processor view or system view, and on which tabbed page it should appear. The example 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.

To display the dialog shown in Figure 5-66, select **Properties...** from the pop-up menu.

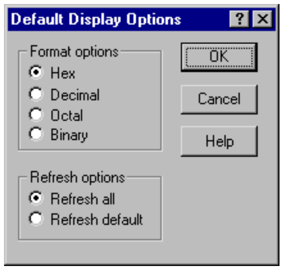


Figure 5-66 Default Display Options dialog

5.5.4 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 5-67, containing two tabbed pages, labeled **RDI Log** and **Debug Log**.

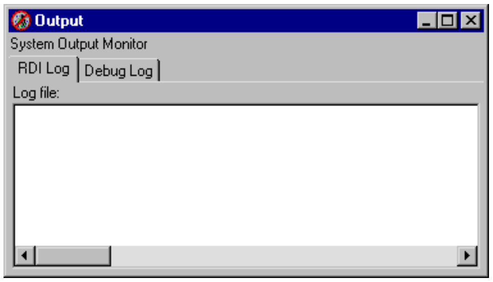


Figure 5-67 Output system view

Click **RDI Log** to see the page that contains a list of function calls made to the RDI.

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 *Watch/Breakpoints...* on page 5-52). 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 5-68, right-click on either the **RDI Log** page or the **Trace** page.

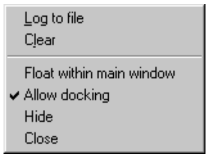


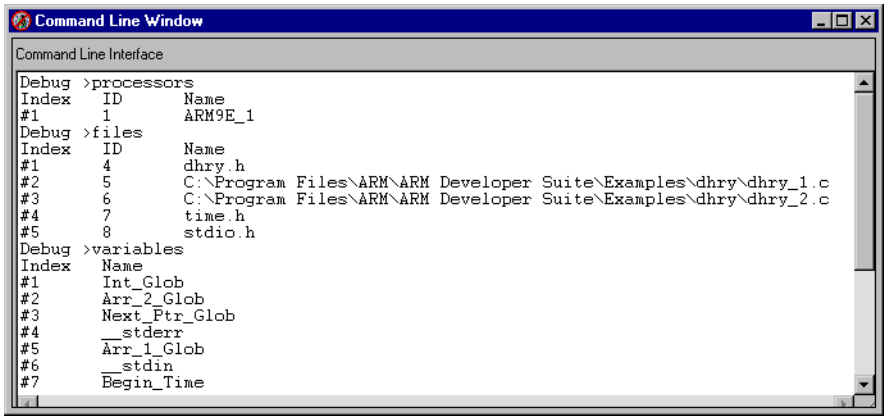
Figure 5-68 Output system view pop-up menu

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.

5.5.5 Command Line Interface system view

The *Command Line Interface* (CLI) system view provides you with an alternative method of issuing commands and viewing data. Everything you can do with the menus and dialogs of the graphical user interface you can also do by entering commands in response to CLI prompts, as shown in Figure 5-69. 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 5-69 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 5-70, right-click within the **Command Line Interface** system view.

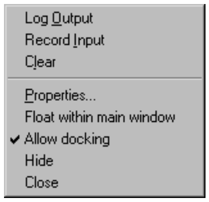


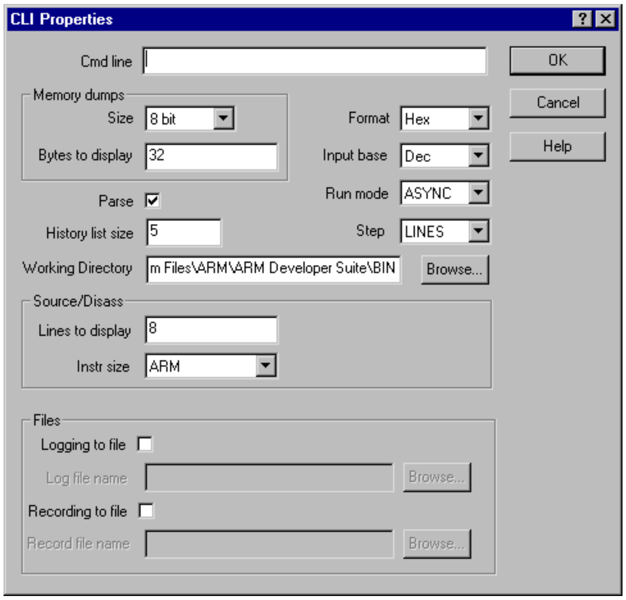
Figure 5-70 CLI system view pop-up menu

Log Output 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 dialog shown in Figure 5-71, select **Properties...** from the pop-up menu.



Cmd line
Memory dumps
Size: 8 bit
Bytes to display: 32
Format: Hex
Input base: Dec
Run mode: ASYNC
Step: LINES
Parse: ✓
History list size: 5
Working Directory: m Files\ARM\ARM Developer Suite\BIN
Source/Disass
Lines to display: 8
Instr size: ARM
Files
Logging to file: ☐
Log file name
Recording to file: ☐
Record file name

Figure 5-71 Command Line Interface Properties dialog

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

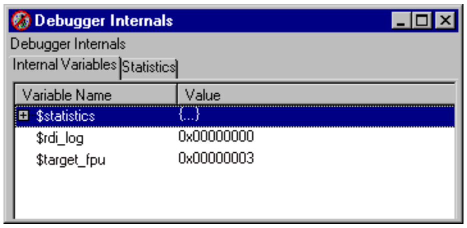
5.5.6 Debugger Internals system view

The Debugger Internals system view has two tabbed pages:

- *Internal Variables*
- *Statistics* on page 5-48.

Internal Variables

The first tabbed page of the Debugger Internals system view shows Internal Variables, as shown in Figure 5-72.



Variable Name	Value
$statistics	{...}
$rdi_log	0x00000000
$target_fpu	0x00000003

Figure 5-72 Debugger Internals - Internal Variables

The debugger, like most programs, uses variables. The following are made available to you 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.

$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).

$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 5-56). This variable is read-only.

$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:

0	specifies that no floating-point code is to be used (`none`)
1	selects software floating-point library with pure-endian doubles (`softVFP`), and is the default setting for images built with ADS tools
2	selects software floating-point library with mixed-endian doubles (`softFPA`)
3	selects hardware Vector Floating-Point unit (`VFP`)
4	selects hardware Floating-Point Accelerator (`FPA`).

SoftVFP and SoftFPA images run correctly on a target whether or not hardware floating point is present, but VFP and FPA images must be run on the appropriate hardware.

You can examine the contents of these variables, and change the values stored in some of them.

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 5-73.

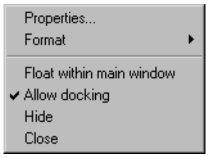
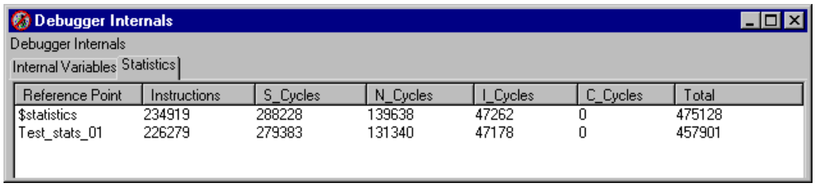


Figure 5-73 Internal Variables pop-up menu

Use this pop-up menu to set properties and to select a display format. Refer to AXD online help for details.

Statistics

The second tabbed page of the Debugger Internals system view shows statistics, as shown in Figure 5-74.



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 5-74 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 labelled `$statistics`.

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.

——— **Note** ———

When emulating 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 5-75.

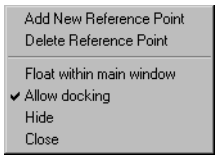


Figure 5-75 Statistics pop-up menu

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.

5.6 Execute menu

The **Execute** menu (see Figure 5-76), lets you control how execution continues from the current point.

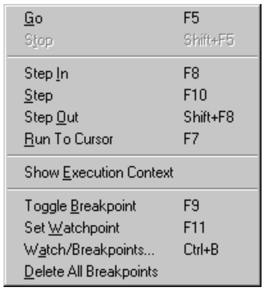


Figure 5-76 Execute menu

The **Execute** menu items are described in the subsections that follow.

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

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 Tools Guide*.

5.6.4 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.

5.6.5 Step Out

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

5.6.6 Run To Cursor

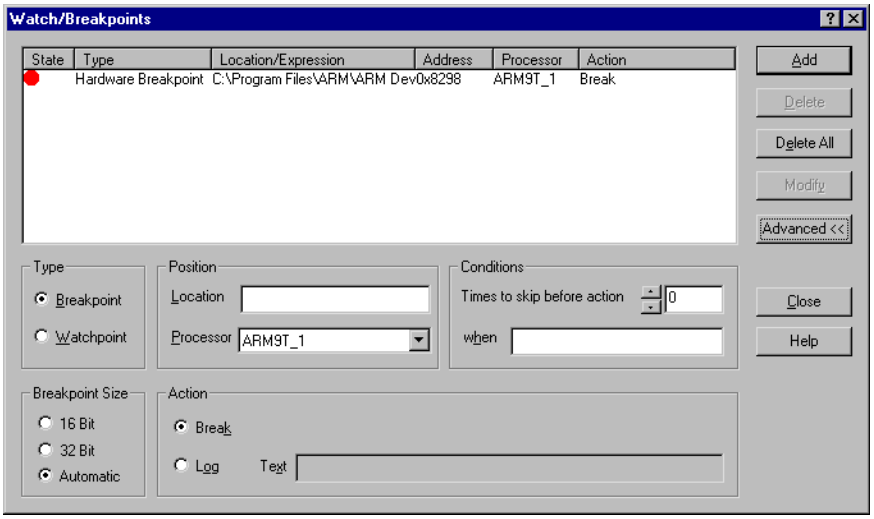
This continues execution but stops when the next instruction to be executed is the one where you have positioned the cursor.

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 Watch/Breakpoints...

This selects **Watch/Breakpoints...** from the **Execute** menu to set, modify, or remove breakpoints or watchpoints. The resulting dialog is shown in Figure 5-77.



State	Type	Location/Expression	Address	Processor	Action
●	Hardware Breakpoint	C:\Program Files\ARM\ARM Dev	0x8298	ARM9T_1	Break

Type: [✓ Breakpoint] Watchpoint

Position: Location: Processor: ARM9T_1

Conditions: Times to skip before action: 0 when:

Breakpoint Size: 16 Bit 32 Bit [✓ Automatic]

Action: [✓ Break] Log Text:

Figure 5-77 Watch/Breakpoints dialog

The Watch/Breakpoints dialog is normally displayed without the Breakpoint Size and Action boxes. These are seldom needed. If you do need these boxes, click the **Advanced>>** button to enlarge the dialog and display them as shown.

The main box of the display shows details of any breakpoints or watchpoints that are currently set. You can select one of these to modify, or to delete, or you can delete all the listed breakpoints and watchpoints, by clicking buttons at the right of the dialog.

To add a new breakpoint or watchpoint, enter the details in the Type, Position, and Conditions boxes. In the Conditions box you can enter `n` in the **Times to skip before action** field to execute normally while the breakpoint is reached (or the watchpoint value changes) `n` times, and stop on the (`n+1`)th time. You can also enter in the **When** field an expression that must evaluate to true before program execution stops. When you are satisfied with all the settings, click the **Add** button.

To modify an existing breakpoint or watchpoint, select it in the display. Its details appear in the Type, Position and Conditions boxes. Change the settings in those boxes as required, then click on the **Modify** button.

To delete a breakpoint or watchpoint, select it in the display then click the **Delete** button.

To delete all the listed breakpoints and watchpoints, click the **Delete All** button.

To disable an existing breakpoint or watchpoint, click the red disk at the left of its line. The centre of the disk becomes grey. Click the disk again to restore normal operation.

In some cases the debugger might not be able to determine whether it is debugging ARM code or Thumb code. For example:

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

If you need to specify the type of code in use, the **Breakpoint Size** buttons are enabled.

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 added to the log 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 5-43). 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.

5.6.9 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.

5.6.10 Set Watchpoint

When you are viewing a source file or a disassembly, you can set or replace a watchpoint on the currently selected item by selecting **Set Watchpoint** from the **Execute** menu.

You can also set or replace a watchpoint by selecting an item in a source file or disassembly view, right-clicking, and selecting **Set Watchpoint** from the pop-up 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 5-78, 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.

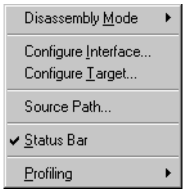


Figure 5-78 Options menu

The **Options** menu items are described in the subsections that follow.

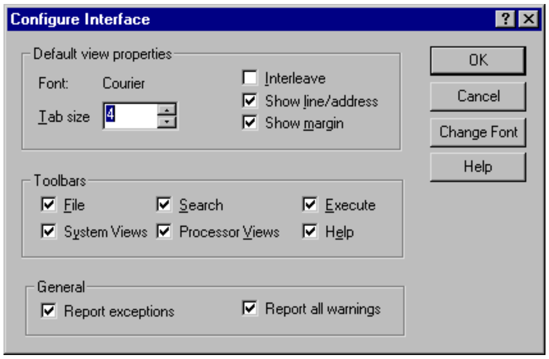
5.7.1 Disassembly Mode

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 may 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 is similar to that shown in Figure 5-79.



Default view properties
Font: Courier
Tab size 4
☐ Interleave
☑ Show line/address
☑ Show margin

Toolbars
☑ File ☑ Search ☑ Execute
☑ System Views ☑ Processor Views ☑ Help

General
☑ Report exceptions ☑ Report all warnings

Figure 5-79 Configure Interface dialog

The **Default view properties** you set in this dialog are used, when applicable, for each view you display. You can also change these settings in any displayed view.

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 5-2.

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 5-43).

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.

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

- *ARMulator configuration*
- *Multi-ICE configuration* on page 5-58
- *BATS configuration* on page 5-60
- *Remote_A configuration* on page 5-61.

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.

To simulate the ARM966E-S core, you need a different version of ARMulator. For this special version, locate and select file `armul9xxe.dll`.

Select the ARMulator target line and click the **Configure** button to display the dialog shown in Figure 5-80.

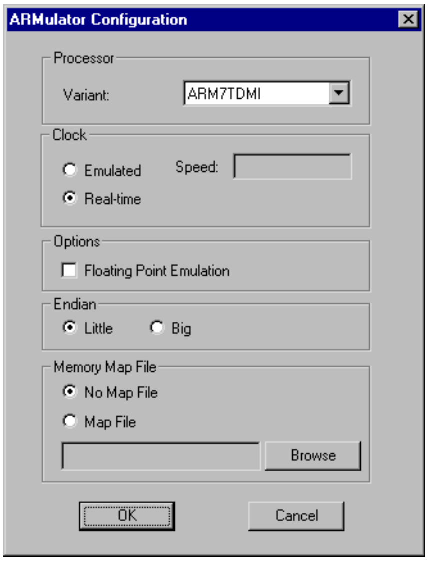


Figure 5-80 ARMulator configuration dialog

The ARMulator configuration dialog enables you to:

- specify which ARM processor you want ARMulator to emulate
- choose between emulating a processor clock running at a speed that you can specify, or executing instructions in real time
- specify whether a floating-point emulator is to be used
- specify that the emulated target is to operate in little-endian or big-endian mode
- 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. 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 `armul.cnf` 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 emulated **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 5-46. 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 `armul.cnf` is used.

For ARMulator, an unspecified clock frequency is of no consequence because ARMulator does not need 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 emulated 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 `$clock`. It is also used internally by ARMulator in the calculation of `$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.

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

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 dialog shown in Figure 5-81.

The Multi-ICE configuration dialog enables you to:

- specify the network address of the computer on which the Multi-ICE Server software is running
- select a processor driver

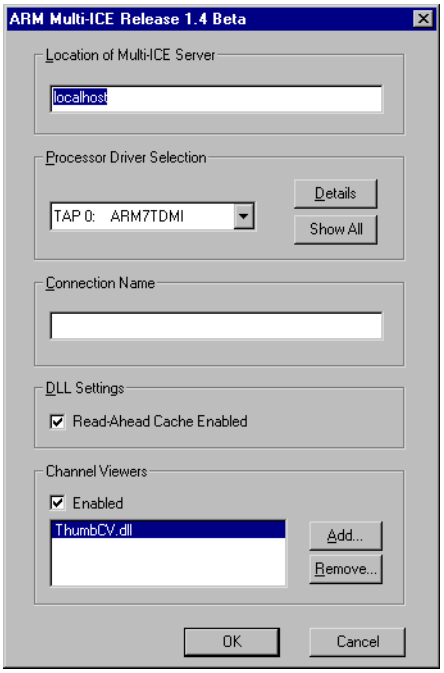
- specify a connection name (required only when access to the Multi-ICE Server software is across a network)
- specify DLL Settings to control the use of various debugger features

  ——— **Note** ———

  The only such feature at present is Read Ahead Cache Enabled. This improves memory read performance by reading more memory than the debugger requests and caching the rest in case it is needed. The DLL learns which regions of memory are safe to access from previous read requests and never reads memory that has not been accessed previously. For certain operations this improves performance considerably, for example stepping with many string variables displayed in a debugger window. The setting is saved and is on by default. If you are debugging a system with demand paged memory, switch this feature off.

  ———————————

- select a channel viewer (check the **Enabled** check box if you want to add viewers to or remove viewers from the list or to select one of the listed viewers).


ARM Multi-ICE Release 1.4 Beta

Location of Multi-ICE Server: localhost

Processor Driver Selection: TAP 0: ARM7TDMI

Connection Name:

DLL Settings

☑ Read-Ahead Cache Enabled

Channel Viewers

☑ Enabled

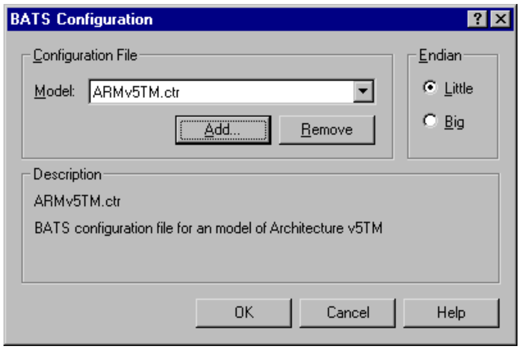
ThumbCV.dll

Figure 5-81 Multi-ICE configuration dialog

BATS configuration

If you need to add BATS to the list of available targets, click **Add** and use the resulting browse dialog to locate and select the `bats.dll` file.

Select the BATS target line and click the **Configure** button to display the dialog shown in Figure 5-82.



Model: ARMv5TM.ctr

Endian: [✓ Little] Big

Description:
ARMv5TM.ctr
BATS configuration file for an model of Architecture v5TM

Figure 5-82 BATS configuration dialog

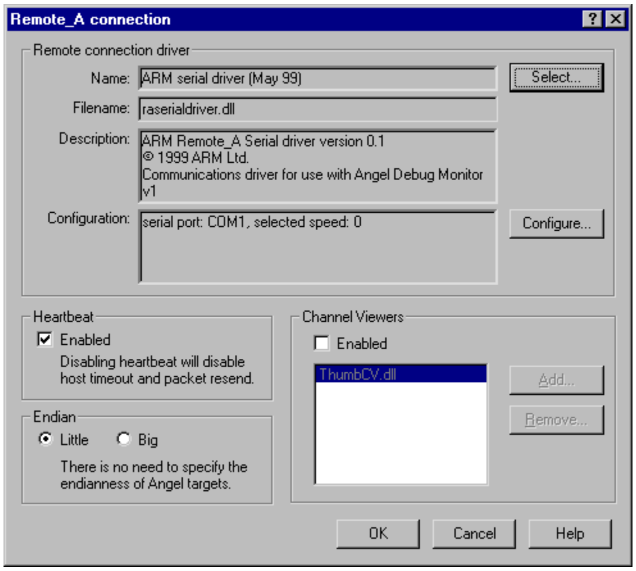
In the BATS configuration dialog you can:

- click **Add** to display a browse dialog and select a filename to add to the (initially empty) list of available configuration files
- click **Remove** to remove the currently displayed filename from the list of available configuration files
- select little-endian or big-endian mode of operation for the target you are preparing to emulate
- click **OK** to configure BATS with the information stored in the configuration file identified by the currently displayed filename
- click **Cancel** to close the dialog without making any change to the current configuration of BATS.

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 5-83.


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.

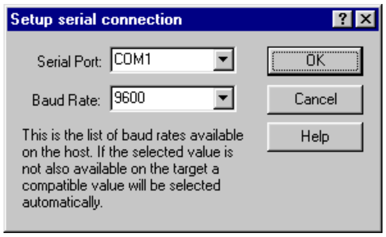
Channel Viewers: Enabled
ThumbCV.dll

Figure 5-83 Configuration of Remote_A connection

The Remote_A connection dialog allows you to examine and/or 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 5-84, Figure 5-85 on page 5-63, or Figure 5-85 on page 5-63.


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 5-84 Serial connection configuration

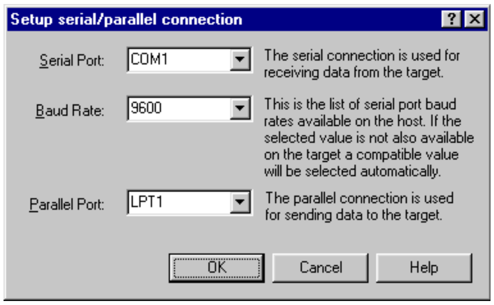

Setup serial/parallel connection
Serial Port: COM1
The serial connection is used for receiving data from the target.
Baud Rate: 9600
This is the list of serial port baud rates available on the host. If the selected value is not also available on the target a compatible value will be selected automatically.
Parallel Port: LPT1
The parallel connection is used for sending data to the target.
OK
Cancel
Help

Figure 5-85 Serial/parallel connection configuration

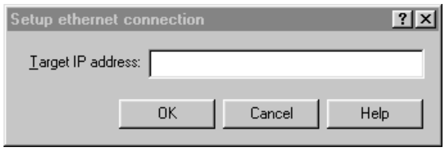


Figure 5-86 Ethernet connection configuration

Heartbeat

Ensures reliable transmission by sending heartbeat messages. If not enabled, there is a danger that the host and the target can get into a deadlock situation, with both waiting for a packet.

Endian

These buttons allow you to inform the debugger that the target is operating in little-endian or big-endian 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 add, remove, or select channel viewers in the displayed list of `.dll` files. See *Comms Channel processor view* on page 5-25 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.

5.7.4 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 5-4).

If you click on the **Status Bar** menu item so that it is cleared the status bar is not displayed.

5.7.5 Profiling

Point to **Profiling** to display a submenu, shown in Figure 5-87. This allows you to control profiling, provided you made suitable settings when you loaded the image. See *Profiling* on page 4-12 for details.

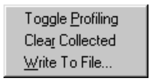
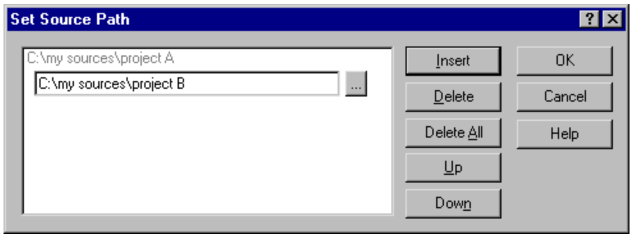


Figure 5-87 Profiling submenu

5.7.6 Source Path...

Select **Source Path...** from the **Options** menu to display the **Set Source Path** dialog shown in Figure 5-88. 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

Figure 5-88 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 by AXD and used in subsequent debugging sessions.

5.8 Window menu

The **Window** menu, shown in Figure 5-89, allows you to control the display of windows and icons on your screen.

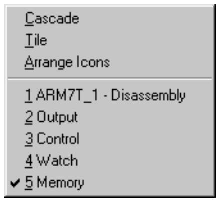


Figure 5-89 Window menu

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 2-9.

The **Window** menu items are described in the subsections that follow.

5.8.1 Cascade

Cascade operates on any windows set to float within the main window. They are repositioned, resized, and overlapped, so as 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

Tile operates on any windows set to float within the main window. They are repositioned and resized so as to avoid any overlapping and to fill the area of the main window that remains unoccupied by any docked windows.

5.8.3 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.

5.8.4 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 5-90, provides you with access to AXD online help and to details of the version of AXD that you are running.

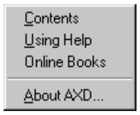


Figure 5-90 Help menu

The **Help** menu items and relevant toolbar icons are described in the subsections that follow.

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. If this option is unavailable, select **Start** → **Programs** → **ARM Developer Suite v1.0** → **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*
- *To automate repetitive tasks*.

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

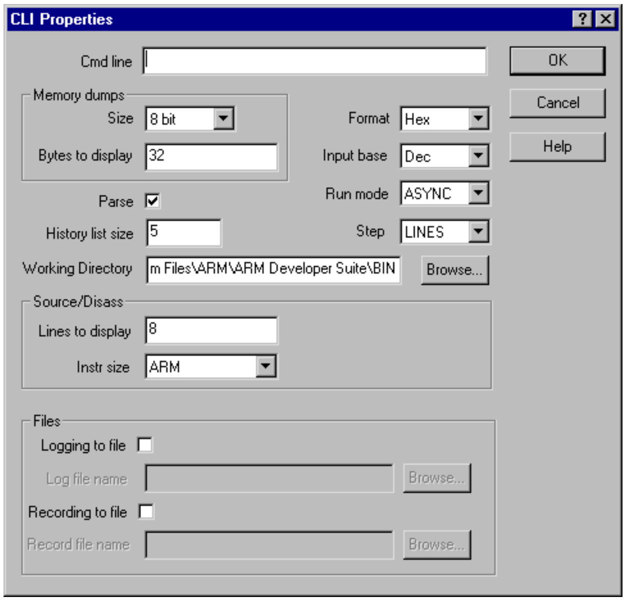
You can record the commands you issue in a log file (see *Command Line Interface system view pop-up menu* on page 5-44 or *record* on page 6-27). You can then easily repeat the same commands by submitting the file to the CLI using the **obey** command (see *obey* on page 6-25).

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 5-44.

To display the *CLI properties dialog* shown in Figure 6-1, select **Properties...** from the pop-up menu.



Cmd line

Memory dumps

Size 8 bit

Bytes to display 32

Format Hex

Input base Dec

Run mode ASYNC

Parse ✓

History list size 5

Step LINES

Working Directory m Files\ARM\ARM Developer Suite\BIN

Source/Disass

Lines to display 8

Instr size ARM

Files

Logging to file ☐

Log file name

Recording to file ☐

Record file name

Figure 6-1 CLI properties dialog

The CLI Properties dialog allows you to set various default values so that you can avoid the need 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** to display more information about this dialog.

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 need to 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

The `@` symbol indicates that the parameter is an address. The remainder of the parameter is an expression that is evaluated to a memory address before use.

6.2.4 $ parameters

The `$` symbol indicates that the parameter is a low-level symbol or an internal debugger symbol.

6.2.5 + parameters

The `+` symbol prefixes the second parameter of a range when it is to be used as a size rather than an upper value.

6.3 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 6-11.

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 *`runmode`* is specified as a parameter, you can enter either the string `ASYNC` or the string `SYNC`. Any other value for this parameter is invalid.

In the alphabetical list of *Commands* on page 6-11, 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 upper-case and lower-case characters. The parameters for which you must specify certain values only are described in the subsections that follow.

6.4.1 runmode

The *`runmode`* parameter must be set to `ASYNC` or `SYNC`.

Setting *`runmode`* to `ASYNC` allows the command-line window to continue functioning while commands are executed. To run a script, each command must execute completely before the next command is issued, and in this case you must set *`runmode`* to `SYNC`.

The setting of *`runmode`* is shown in the **Run mode** field of the CLI Properties dialog.

6.4.2 format

The *`format`* parameter must be set to `Hex`, `Dec`, `Oct`, or `Bin`.

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 determines whether each value is displayed in hexadecimal, decimal, octal or binary format.

You can also use the *`format`* parameter to specify the display format for registers.

The setting of *`format`* is shown in the **Format** field of the CLI Properties dialog.

6.4.3 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 should make the setting itself (auto). You generally need to specify the instruction type if the code was built without debug information.

The setting of *`asm`* is shown in the **Instr size** field of the CLI Properties dialog.

6.4.4 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`).

The setting of *`instr`* is shown in the **Step** field of the CLI Properties dialog.

6.4.5 step

The *`step`* parameter must be set to `IN` or `OUT`.

This affects the way an instruction calling a function is processed. `IN` specifies that the step proceeds only to the first executable instruction in the called function. `OUT` specifies that the step includes execution of the called function and proceeds to the instruction at which execution returns to the calling program.

6.4.6 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.

6.4.7 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.

6.4.8 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 need 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.

`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 (the current processor is assumed if you do not specify a processor).

`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.

`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 (the image associated with the current processor is assumed if you do not specify an image).

`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 #.

`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 which 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 such objects. Specify the index as a value prefixed by `#`.

`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.
`breakpoint`	You specify a breakpoint as its index in the breakpoint list, in the form of a value prefixed by `#`.
`watchpoint`	You specify a watchpoint as its index in the watchpoint list, in the form of a value prefixed by `#`.
`runmode`	You must specify this as either `SYNC` or `ASYNC`.
`step`	This controls the amount of processing that takes place following an instruction that calls a function. You must specify this as either `In` or `Out`.
`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.
`regbank`	You can identify a register bank by: • the name of the register bank • 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 as shown in the output of a registerbanks command.
`memory`	Denotes that memory is to be displayed in bytes, halfwords, or words. You must specify `8`, `16`, or `32`.
`format`	Denotes the format in which the contents of memory, registers, or variables are displayed. You must specify `Hex`, `Dec`, `Oct`, or `Bin`.
`scope`	Denotes which context variables to display, based on their scope. You must specify `Class`, `Local`, or `Global`.
`toggle`	Where this parameter is allowed, you can use it to switch on or off certain properties. You must specify either `On` or `Off`.
`asm`	Denotes that assember 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.
`register`	You can identify a register by: • the name of the register • the index of the register in the current register list, in the form of a value prefixed with `#`.
`value`	You specify a numeric value.
`string`	You specify a text string enclosed in quotes ("...").

ipvariable

Denotes any one of a group of variables that define image-related properties. The ipvariables 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 ("...").

ppvariable

Denotes any one of a group of variables that define processor-related properties. The ppvariables currently supported are:

`vector_catch`

Defines which exceptions in the processor are intercepted by the debugger. See *Processor pop-up menu* on page 5-35 for more information.

`comms_channel`

Enables (1) or disables (0) the debug communications channel viewer. If this option is enabled you cannot use DCC semihosting.

`semihosting_enabled`

Enables or disables semihosting. Valid values are:

`0`	Semihosting is disabled.
`1`	Standard semihosting is enabled.
`2`	DCC semihosting is enabled. This applies to Multi-ICE only.

`semihosting_vector`

Contains the address of the breakpoint used to detect a semihosted SWI (0x8 by default). See the description of the semihosting SWIs in the *ADS Debug Target Guide* for more information.

`semihosting_dcchandler_address`

Contains the address in memory where Multi-ICE will place its DCC handler code. See the *Multi-ICE User Guide* for more information.

`arm_semihosting_swi`

Defines the ARM software interrupt number reserved for semihosting. You should not normally change this.

`thumb_semihosting_swi`

Defines Thumb software interrupt number reserved for semihosting. You should not normally change this.

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 6-8 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.

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.

Where lines of output are described, `<tab>` indicates the presence of a tab character.

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

The `addsourcedir` command inserts the specified directory in the current list of paths at the specified position. The paths in this list, and the order in which they are listed, specify the paths on local or remote machines searched when a source file is required.

The shorthand form of the `addsourcedir` command is `asd`.

Syntax

```
asd path[ index]
```

where:

`path`	is a fully qualified directory, or a list of fully qualified directories separated by `;` for Windows or `:` for UNIX. Enclose the path specification in quotes if it contains spaces.

index	specifies the position of the new entry in the list. A value of 0 or 1 places the new entry at the head of the list. A value greater than the current list size or unspecified places the new entry at the end of the list. If no index is specified, the path is added to the end of the list.

Example

```
asd "c:\my sources\project B" 3
```

6.6.2 backtrace

See *stackentries* on page 6-35.

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, in the following format:

```
index<tab>{file}:line<tab>@address<tab>SW|HW
```

The position in this list, `index`, gives you a convenient way of referring to a breakpoint in such commands as `clearbreak`.

If you supply parameters, the command creates and sets a new simple breakpoint so that execution continues while the specified address is visited *skip* times and stops on the (*skip*+1)th time. If you do not specify a value for *skip*, a default value of 0 is assumed, so execution stops every time the address is visited.

The shorthand form of the `break` command is `br`.

Syntax

```
br[ expr|position [, skip]]
```

where:

expr|position	is either an expression or a position that defines where a new breakpoint is to be created.
skip	specifies the number of times an arrival at the breakpoint is ignored before an arrival is acted upon.

Examples

`br c:\test\main.c|130 100`
: sets a breakpoint on line 130 of file main.c, with 100 arrivals ignored before execution is interrupted.

`br #5|150` sets a breakpoint at line 150 of file number 5. The index #5 must have been obtained using the `files` command.

——— **Note** ———

To set complex breakpoints, select **Watch/Breakpoints...** from the **Execute** 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.

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

Syntax

```
clear
```

6.6.8 clearbreak

The `clearbreak` command unsets and deletes the specified breakpoint. See *break* on page 6-12 for a description of how to refer to a breakpoint.

The shorthand form of the `ClearBreak` command is `cbr`.

Alias

`unbreak` is an alternative to `clearbreak`.

Syntax

```
cbr breakpoint
```

Examples

`cbr #2`	clears breakpoint number 2. The index #2 must have been obtained using the `break` command.
`unbreak #2`	has exactly the same effect.

6.6.9 clearwatch

The `clearwatch` command unsets and deletes the specified watchpoint. See *watchpt* on page 6-38 for a description of how to refer to a watchpoint.

The shorthand form of the `clearwatch` command is `cwpt`.

Alias

`unwatch` is an alternative to `clearwatch`.

Syntax

```
cwpt watchpoint
```

Examples

`cwpt #2`	clears watchpoint number 2. The index #2 must have been obtained using the `watchpt` command.
`unwatch #2`	has exactly the same effect.

6.6.10 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
```

6.6.11 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 6-35 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 have been obtained using the `stackentries` command.

6.6.12 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 6-35).
`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 have been obtained with the `stackentries` command.

`convar local`
: displays the local variables in the current context, in hexadecimal format.

6.6.13 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 have been obtained using the `classes` command.

6.6.14 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:

```
index<tab>variablename<tab>value
```

The shorthand form of the `dbginternals` command is `di`.

Syntax

```
di
```

6.6.15 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 alternative to `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.
`asm`	can be set to `ARM`, `Thumb`, or `Auto` (see *asm* on page 6-6). If not specified, the current value of the appropriate internal CLI variable is used.

6.6.16 examine

See *memory* on page 6-24.

6.6.17 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]
```

6.6.18 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 the area 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).

6.6.19 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 at which you want the search to begin.
high-expr	is an expression that evaluates to the memory address at which you want the search to end.

6.6.20 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 at which you want the search to begin.
high-expr	is an expression that evaluates to the memory address at which you want the search to end.

6.6.21 format

The `format` command sets the default formats for both input and output of simple values.

The shorthand form of the `format` command is `fmt`.

Syntax

```
fmt inbase, outformat
```

where:

inbase	can be set to `Hex`, `Dec`, or `Oct`.
outformat	can be set to `Hex`, `Dec`, `Oct`, or `Bin` (see *format* on page 6-6).

6.6.22 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<tab><tab>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]
```

6.6.23 getfile

See *loadbinary* on page 6-23.

6.6.24 go

See *run* on page 6-28.

6.6.25 help

The `help` command invokes AXD online help.

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

Syntax

```
hlp
```

6.6.26 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 6-28.

The shorthand form of the `images` command is `im`.

Syntax

```
im
```

6.6.27 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. 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`*`]`

6.6.28 let

See *setwatch* on page 6-34.

6.6.29 list

See *disassemble* on page 6-18.

6.6.30 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()`.

6.6.31 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 alternative to `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.

6.6.32 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 alternative to `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.

6.6.33 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 6-27.

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

Syntax

```
log[ file]
```

6.6.34 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><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]
```

6.6.35 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`.

Alias

`examine` is an alternative to `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.
memory	can be set to `8`, `16`, or `32` (see *memory* on page 6-7).
format	can be set to `Hex`, `Dec`, `Oct`, or `Bin` (see *format* on page 6-6).

6.6.36 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.

6.6.37 parse

The `parse` command sets the parsing state on or off according to the supplied parameter.

You should normally leave `parse` set to its default value of `ON` so that commands are checked for valid syntax. Set `parse OFF` only when you use the CLI `script` command which uses the CLI expansion mechanism to generate new CLI commands.

The shorthand form of the `Parse` command is `par`.

Syntax

```
par toggle
```

where:

`toggle`	must be set to `ON` or `OFF`.

6.6.38 print

See *watch* on page 6-38.

6.6.39 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 6-37 and *run* on page 6-28.

The shorthand form of the `processors` command is `proc`.

Syntax

```
proc
```

6.6.40 procproperties

The `procproperties` command displays internal variables related to the specified processor, or to the current processor if you do not specify a processor. 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]
```

6.6.41 putfile

See *savebinary* on page 6-30.

6.6.42 quitdebugger

The `quitdebugger` command ends execution of AXD.

The shorthand form of the `quitdebugger` command is `quitd`.

Syntax

```
quitd
```

6.6.43 readsyms

See *loadsymbols* on page 6-23.

6.6.44 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 6-24.

The shorthand form of the `Record` command is `rec`.

Syntax

```
rec[ file]
```

6.6.45 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.

The shorthand form of the `regbanks` command is `regbk`.

Syntax

```
regbk[ processor]
```

6.6.46 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 6-33, 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 name `Current` is listed. See *regbanks* 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).

6.6.47 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 6-21 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.

6.6.48 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 alternative to `run`.

Syntax

```
r[ processor][, runmode]
```

where:

processor	specifies the processor (the current processor is the default)
runmode	if specified must be set to `ASYNC` or `SYNC` (see *runmode* on page 6-6).

6.6.49 runmode

The `runmode` command allows you to examine or set the mode of execution and the step size. To see the current settings, issue the command with no parameters.

The shorthand form of the `runmode` command is `rmode`.

Syntax

```
rmode[ runmode[, instr]]
```

where:

runmode	can be set to `SYNC` or `ASYNC` (see *runmode* on page 6-6).
instr	can be set to `INSTR` or `LINE` (see *instr* on page 6-7), but is overridden if no source is available.

6.6.50 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 `RunToPosition` command is `rto`.

Syntax

```
rto position[, processor]
```

where:

position	is an expression that evaluates to a memory address.
processor	identifies the processor.

6.6.51 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 alternative to `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.

6.6.52 script

Not available in ADS v1.0.

6.6.53 setimgprop

The `setimgprop` command sets an image-related internal variable to the specified value (see *imgproperties* on page 6-22). 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 6-8 for a list of valid ipvariables.
value	specifies the new value to be assigned to the specified variable.

6.6.54 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).

6.6.55 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
```

6.6.56 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
```

6.6.57 setprocprop

The `setprocprop` command sets a processor-related internal variable to the specified value (see *procproperties* on page 6-26). You need 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
```

where:

ppvariable
: specifies the ppvariable to be reset. See *Definitions* on page 6-8 for a list of valid ppvariables.

value
: specifies the new value to be assigned to the specified variable.

6.6.58 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 6-28). If you do not specify a register bank, the register bank named `Current` is used.

The expression can also identify a register and include a register bank name.

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`.

6.6.59 setsourcedir

The `setsourcedir` command sets the list of paths to that specified.

The paths in this list, and the order in which they are listed, specify the paths searched when a source file is required.

The shorthand form of the `setsourcedir` command is `ssd`.

Syntax

ssd *list of directories*

The list is of fully qualified directory names. Enclose the list in quotes if it contains any spaces. To clear the current list, specify an empty list (`""`) with no space characters between the quotation marks. If you are specifying multiple paths, separate them with '`;`' for Windows NT or '`:`' for UNIX.

Example

```
ssd "c:\my srcs\proj A;d:\proj B;c:\srclib"
```

6.6.60 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 6-38).

The shorthand form of the `setwatch` command is `swat`.

Alias

`let` is an alternative to `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`.

6.6.61 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 alternative to `source`.

Syntax

```
src value1, [+]value2[, file]
```

where:

`value1`	specifies the line number of the source file at which you want the listing to begin.
`value2`	specifies either the line number of the source file at which you want the listing to end 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).

6.6.62 sourcedir

The `sourcedir` command lists the paths searched when a source file is required, in the following format:

```
index<tab>fully qualified directory name
```

The paths are searched in the order in which they are listed.

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

Syntax

```
sdir
```

6.6.63 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 6-16 and *context* on page 6-15, for example.

The shorthand form of the `stackentries` command is `stk`.

Alias

`backtrace` is an alternative to `stackentries`.

Syntax

```
stk[ count]
```

where:

`count`	specifies the number of lines you want listed if you do not want the whole stack displayed.

6.6.64 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
```

6.6.65 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
```

6.6.66 step

The `step` command causes execution to proceed by one step according to the current run mode.

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`. If a subroutine call is encountered, this command steps over it.

6.6.67 stop

The `stop` command stops execution of the specified processor, or of the current processor if you supply no parameter.

The `stop` command is obeyed only when `runmode` is set to its default value of `ASYNC` (see *runmode* on page 6-6).

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

Syntax

```
stop[ processor]
```

6.6.68 type

See *source* on page 6-34.

6.6.69 unbreak

See *clearbreak* on page 6-14.

6.6.70 unwatch

See *clearwatch* on page 6-15.

6.6.71 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]
```

6.6.72 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 6-34.

The shorthand form of the `Watch` command is `wat`.

Alias

`print` is an alternative to `watch`.

Syntax

```
wat expr[, format]
```

6.6.73 watchpt

If you supply parameters, the `watchpt` command creates and sets a new watchpoint so that execution continues normally while the value stored at the specified location changes *`skip`* times, stopping on the `(`*`skip`*`+1)` th time. If you do not specify *`skip`* it takes a default value of 0.

To set complex watchpoints, select **Watch/Breakpoints...** from the **Execute** menu.

If you supply no parameters, the `watchpt` command lists all the watchpoints that are currently set. Each watchpoint is listed on a separate line, in the following format:

```
index<tab>SW|HW<tab>address
```

The position in this list, `index`, gives you a convenient way of referring to a watchpoint in such commands as `ClearWatch`.

The shorthand form of the `WatchPoint` command is `wpt`.

Syntax

```
wpt[ expr[, skip]]
```

6.6.74 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 6-35).

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.

This chapter 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 7-6.

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 version only, this is indicated in the text or in the section heading.

7.1.1 Minimum requirements for UNIX

You can run ADU on either a Sun workstation or an HP workstation, provided they meet the minimum requirements described in the subsections that follow.

Sun workstation

The minimum hardware requirements for installing and running ADU are:

- Sun UltraSparc or compatible machine
- CD-ROM drive (this can be a networked CD-ROM drive).

The minimum software requirements for installing and running ADU are:

- Solaris 2.5.1 or 2.6, with the *Common Desktop Environment* (CDE)
- ARM Developer Suite v1.0.

HP workstation

The minimum hardware requirements for installing and running ADU are:

- HP PA-RISC machine
- CD-ROM drive (this can be a networked CD-ROM drive).

The minimum software requirements for installing and running ADU are:

- HP-UX 10.20
- ARM Developer Suite v1.0.

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.

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*, which can be:
  - hardware-based on an ARM core
  - software that emulates an ARM core.
- a *debugger,* such as ADW, ADU, or armsd.

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

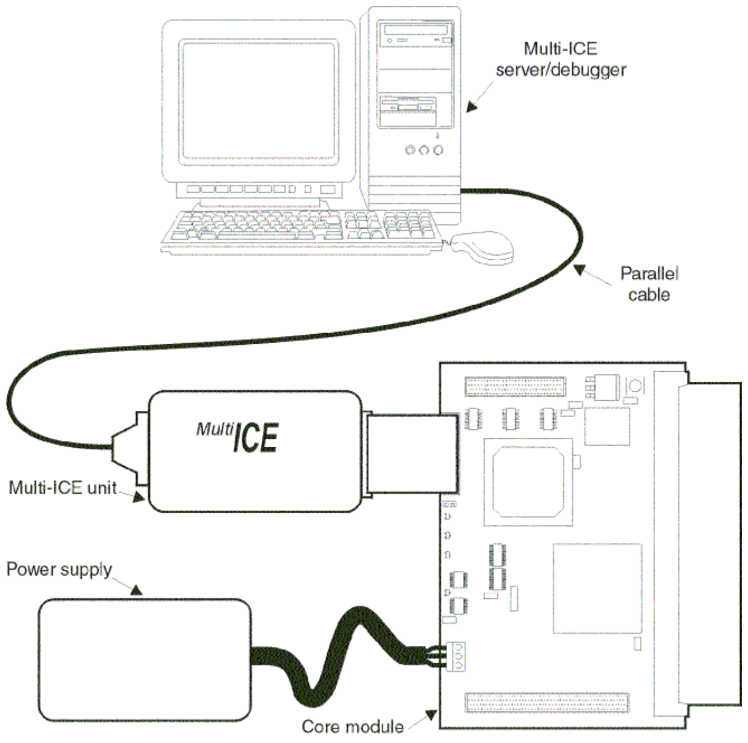


Figure 7-1 A typical debugging set-up

7.4 Debugging systems

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

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

7.4.1 ARMulator

ARMulator is a collection of programs that emulate 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 *Configurations* on page 9-25. For detailed information on Multi-ICE refer to the Multi-ICE documentation.

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.

7.5 Debugger concepts

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

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

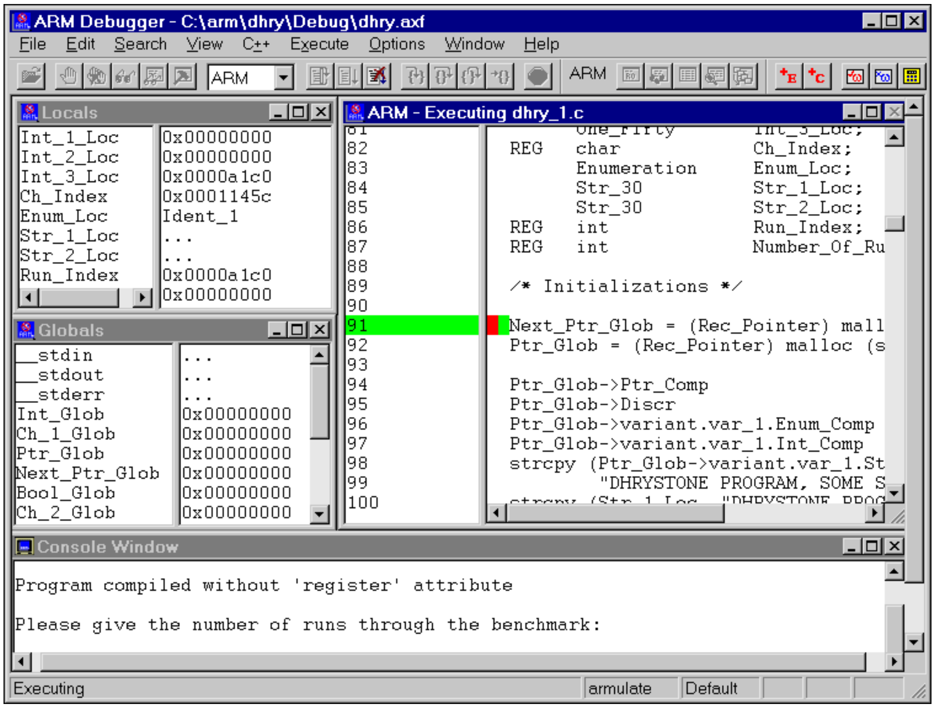
This chapter describes the main features of the ADW and ADU desktop and gives you enough information to start working with the debugger. Additional features are described in Chapter 9 *Working with ADW and ADU*. This chapter contains the following sections:

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*.
- 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 Locals windows, in the process of debugging the sample image `DHRY`.


ARM Debugger - C:\arm\dhry\Debug\dhry.axf
File Edit Search View C++ Execute Options Window Help
ARM
ARM
Locals
ARM - Executing dhry_1.c
Globals
Console Window

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

__stdin	...
__stdout	...
__stderr	...
Int_Glob	0x00000000
Ch_1_Glob	0x00000000
Ptr_Glob	0x00000000
Next_Ptr_Glob	0x00000000
Bool_Glob	0x00000000
Ch_2_Glob	0x00000000

```
        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_Ru

/* Initializations */

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

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.St
           "DHRYSTONE PROGRAM, SOME S
strcpy (Str_1_Loc, "DHRYSTONE PROG
```

```
Program compiled without 'register' attribute

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

Executing | armulate | Default

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.

The ARM C++ compiler supplied as part of ADS adds extra features to ADW and ADU. In particular, a **C++** menu appears between the **View** and **Execute** menus. This provides options relevant only to C++ program debugging. C++ also adds its own mini toolbar. See Chapter 10 *Using ADW and ADU with C++* for more information.

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*
- *Starting ADU*
- *ADW and ADU arguments*
- *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 **ARM Developer Suite** program folder or select **Start** → **Programs** → **ARM Developer Suite v1.0** → **ADW Debugger**
- if you are running Windows NT4, double-click on the **adw.exe** icon in the **ARM Developer Suite\Bin** Program group or select **Start** → **Programs** → **ARM Developer Suite v1.0** → **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, **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 9-20 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, reloading, and executing 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.

8.3.2 Reloading an image

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

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

8.3.3 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 9-9).

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.

———————————

8.4 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 9-19 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 8-11
- *Console window* on page 8-12
- *Command window* on page 8-13.

The can also use the **View** menu to display the:

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

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

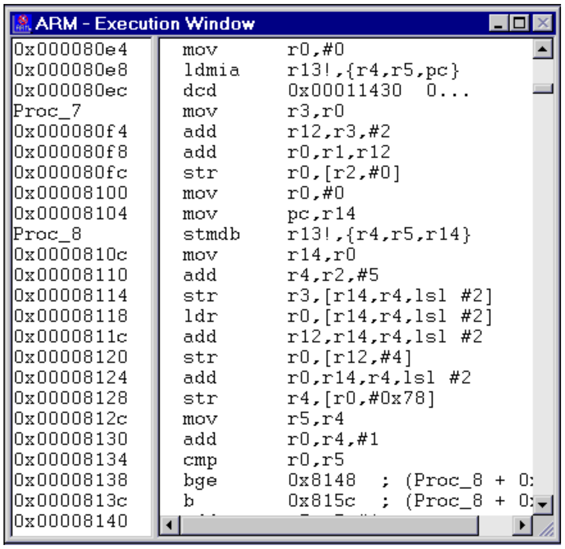
Each of the ADW and ADU desktop windows displays a window-specific menu when you click the secondary mouse button over the window. The secondary button is typically the right mouse button. To activate an item-specific option you must position the cursor over the item in the window before clicking.

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 on page 8-12) displays the source code of the currently executing program.



```
0x000080e4    mov      r0,#0
0x000080e8    ldmia    r13!,{r4,r5,pc}
0x000080ec    dcd      0x00011430  0...
Proc_7        mov      r3,r0
0x000080f4    add      r12,r3,#2
0x000080f8    add      r0,r1,r12
0x000080fc    str      r0,[r2,#0]
0x00008100    mov      r0,#0
0x00008104    mov      pc,r14
Proc_8        stmdb    r13!,{r4,r5,r14}
0x0000810c    mov      r14,r0
0x00008110    add      r4,r2,#5
0x00008114    str      r3,[r14,r4,lsl #2]
0x00008118    ldr      r0,[r14,r4,lsl #2]
0x0000811c    add      r12,r14,r4,lsl #2
0x00008120    str      r0,[r12,#4]
0x00008124    add      r0,r14,r4,lsl #2
0x00008128    str      r4,[r0,#0x78]
0x0000812c    mov      r5,r4
0x00008130    add      r0,r4,#1
0x00008134    cmp      r0,r5
0x00008138    bge      0x8148  ; (Proc_8 + 0:
0x0000813c    b        0x815c  ; (Proc_8 + 0:
0x00008140
```

Figure 8-2 Execution window

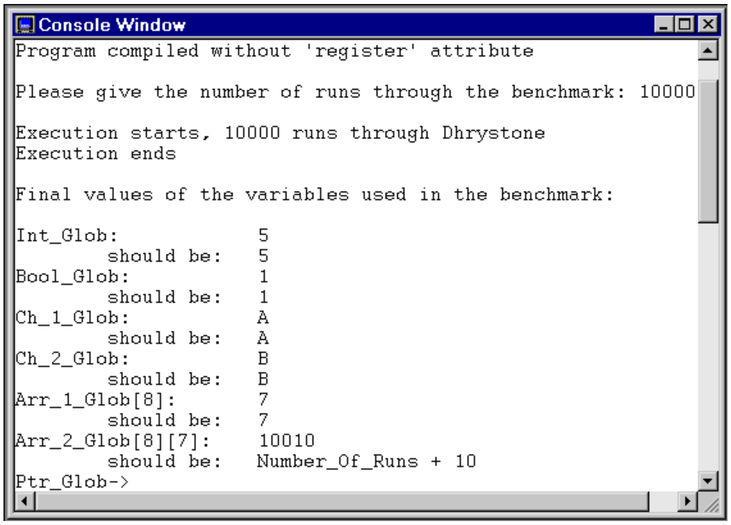
Use the Execution window to:

- 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 on page 8-13) 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

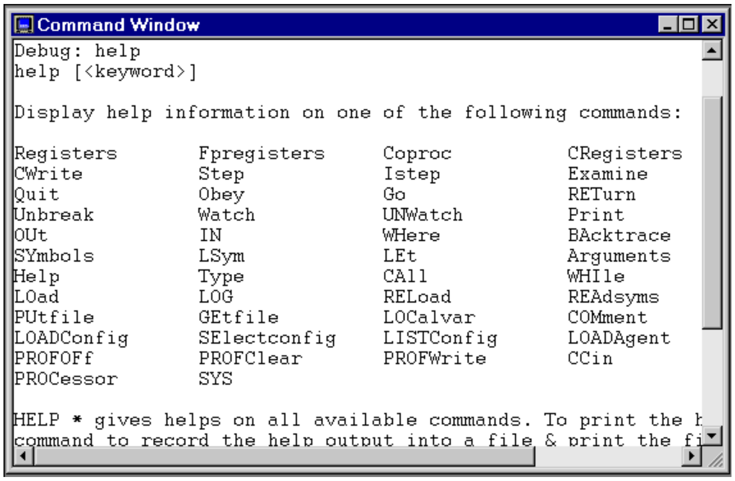
Initially the Console window displays the startup messages of your target processor, for example ARMulator or ARM Development board.

Note

When the executing image needs 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 on page 8-14) 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 9-21 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/disassembled code
- edit or remove breakpoints.

To set a new breakpoint, see *Source File window* on page 8-18.

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 9-12.

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 9-15.

8.5.8 Expression window

The Expression window displays the values of selected variables and/or 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 9-12.

8.5.9 Function Names window

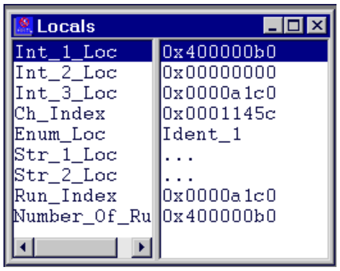
The Function Names window lists the functions that are part of your program.

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 9-12.

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/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 9-16).

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.

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 9-11.

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 9-6
- *Backtrace* on page 9-9
- *Stepping through an image* on page 9-9.

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 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 demonstrated 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.

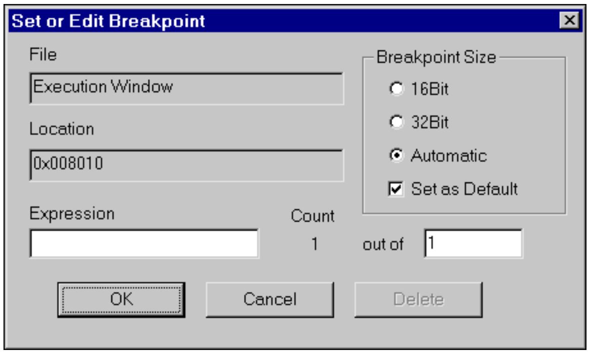
To set a simple breakpoint on a function:

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 (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).

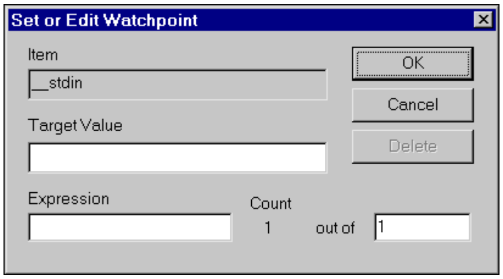


Figure 9-2 Set or Edit Watchpoint dialog

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 would show only the pc 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 in four ways.

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, which 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.

——— **Note** ———

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*
- *Working with variables* on page 9-12
- *Displaying disassembled and interleaved code* on page 9-15
- *Remote debug information* on page 9-16
- *Using regular expressions* on page 9-16
- *High-level and low-level symbols* on page 9-17
- *Profiling* on page 9-18
- *Saving an area of memory to disk* on page 9-19
- *Loading an area of memory from disk* on page 9-19
- *Specifying command-line arguments for your program* on page 9-20
- *Using command-line debugger instructions* on page 9-21
- *Changing the data width for reads and writes* on page 9-21
- *Flash download* on page 9-22.

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 lists the valid format descriptors.
3. Click **OK**.

Table 9-1 Display formats

Type	Format	Description
`int`	Only use this if the expression being printed yields an integer:	
	`%d`	Signed decimal integer (default for integers).
	`%u`	Unsigned integer.
	`%x`	Hexadecimal (lowercase letters).
`char`	Only use this if the expression being printed yields a char:	
	`%c`	Character.
`char*`	`%s`	Pointer to character. Only use this for expressions that yield a pointer to a null terminated string.
`void*`	`%p`	Pointer (`0x%.8lx`), for example, `0x00018abc`. This is safe with any kind of pointer.
`float`	Only use this 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`.

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.

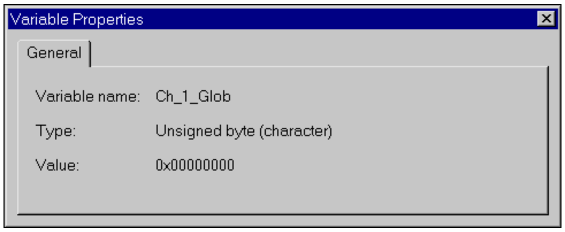
The initial display format of a variable declared as `char[]=` is special. The whole string is displayed, whereas normally arrays are displayed as ellipses (…). If the format is changed it reverts to the standard array representation.

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* (RDI) 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 9-17 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` would match `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. Note that `.` 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]` would match strings `r2` and `r3`. The expression `[a-z]` would match 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 can refer to the same address. Any code between the addresses referred to by the low-level and high-level symbols generally concerns the stack backtrace structure in procedures that conform to the appropriate variants of the *ARM/Thumb Procedure Call Standard* (ATPCS), or argument lists in other procedures. You can display a list of the low-level symbols in your program in the Low Level Symbols window.

In a regular expression, indicate high-level and low-level symbols as follows:

- precede the symbol with `@` to indicate a low-level symbol
- precede the symbol with `^` to indicate a high-level symbol.

9.2.7 Profiling

Profiling involves sampling the *program counter* (pc) at specific time intervals. From this information the percentage of time spent in each procedure can be estimated. 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 Tools 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 on page 9-19).

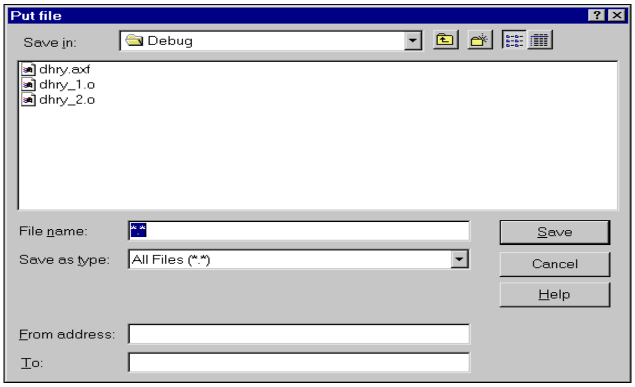


Figure 9-4 Put File dialog

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).

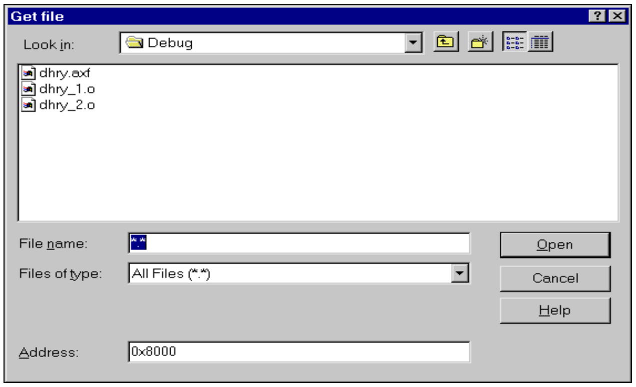


Figure 9-5 Get File dialog

2. Select the file you want to load into memory.

3. Enter a memory address where the file should 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).

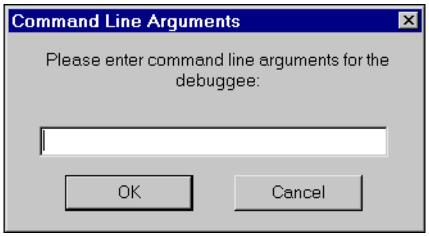


Figure 9-6 Command Line Arguments dialog

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
```

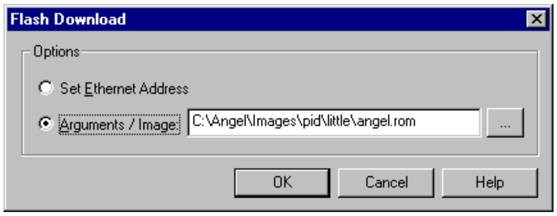
You can also select **View** → **Variables** → **Expression** to open the View Expression window, and use that to specify bytes or shorts for displaying memory. For example, for bytes, enter `*(char *) 0x`*`address`* in the **View Expression** box, and for halfwords, enter `*(short *) 0x`*`address`* 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 Tools Guide*.

9.3 Channel viewers

ADW supports the use of Channel Viewers to access debug communication channels. An example channel viewer is supplied with ADW (`ThumbCV.dll`) or you can provide your own viewer.

Note

ADU also supports the use of Channel Viewers, but ADS does not yet include a channel viewer that runs under UNIX.

9.3.1 ThumbCV channel viewer

To select a 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 the **Channel Viewer Enabled** option. The **Add** and **Remove** buttons are activated.
5. Click the **Add** button and a list of `.DLLs` is displayed.
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 channel viewer. See *Remote_A connection* on page 9-34 for more information on the Remote_A Connection dialog. `ThumbCV.DLL` provides the viewer shown in Figure 9-8.

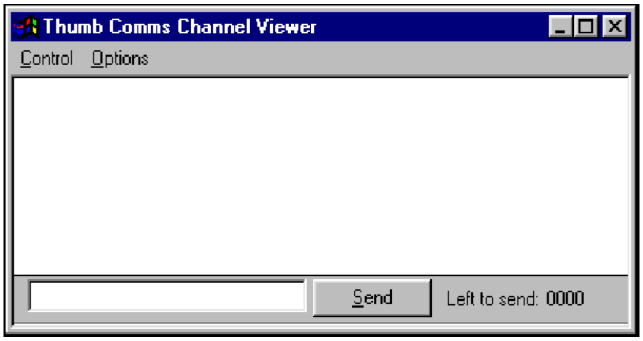


Figure 9-8 Thumb Comms Channel Viewer

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 will store 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 Configurations

You can examine and change the configuration of:

- the Debugger, which includes configuration of:
  - — the target environment for the image being debugged
  - — debugger parameters
  - — startup parameters.
- ARMulator
- a Remote_A connection to Angel or EmbeddedICE
- Multi-ICE
- BATS
- EmbeddedICE.

9.4.1 Debugger configuration

The Debugger Configuration dialog consists of three tabbed screens:

- *Target environment*
- *Debugger* on page 9-26
- *Memory Maps* on page 9-28 (for SDT 2.xx versions of ARMulator only).

Select **Configure Debugger** from the **Options** menu to open the Debugger Configuration dialog.

Target environment

To configure the target environment:

1. Click the **Target** tab of the Debugger Configuration dialog (see Figure 9-9 on page 9-26).
2. Change the following configuration options, as required:

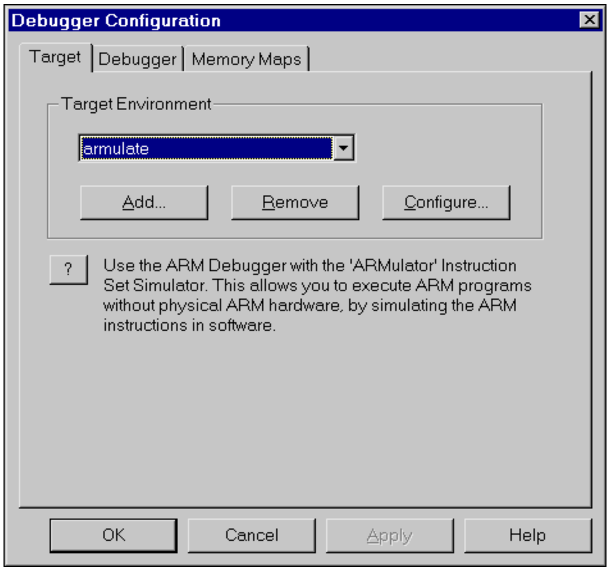
   Target Environment
   Select the target environment for the image being debugged.

   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.

   ? Display a more detailed description of the selected environment.



Target Environment: armulate

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

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.

———————————

Debugger

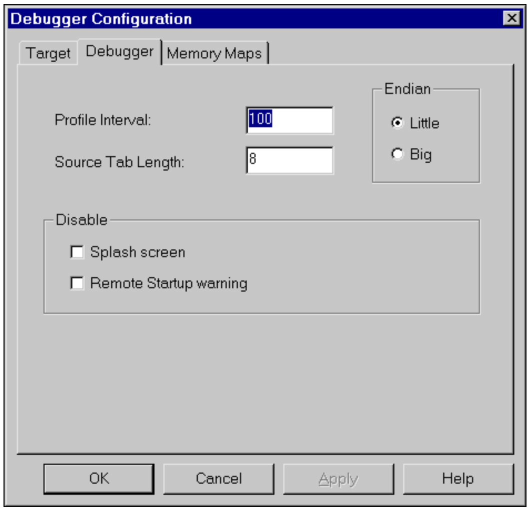
To change the configuration used by the debugger:

1. Click the **Debugger** tab of the Debugger Configuration dialog (see Figure 9-10 on page 9-27).

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.



Profile Interval: 100
Source Tab Length: 8
Endian: (● Little) (○ Big)
Disable
☐ Splash screen
☐ Remote Startup warning

Figure 9-10 Configuration of debugger

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, and 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 Remote_A enabled. 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 in ARMulate.

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.

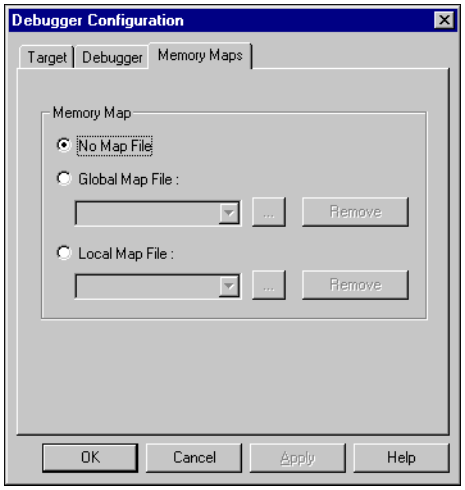
———————————

Memory Maps

For the version of ARMulator that is supplied as part of the ARM Developer Suite you can select a memory map file in the ARMulator Configuration dialog (see *ARMulator configuration* on page 9-31).

For older versions of ARMulator only (for example, the version supplied with SDT 2.50), the matching older version of 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).



Memory Map: [✓ No Map File] Global Map File : Local Map File :

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.

9.4.2 ARMulator configuration

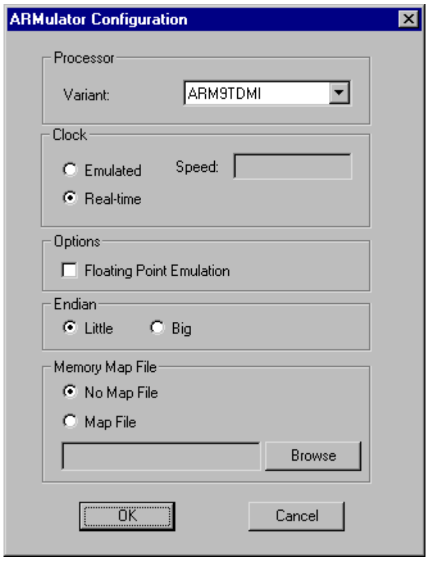
Use the ARMulator Configuration dialog to change configuration settings for ARMulator:

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, and the one that is appropriate for the ARMulator you are using is displayed. Descriptions follow of both ARMulation Configuration dialogs. Be sure to read only the one you need. When you are satisfied with all the settings, click **OK**.

Configuration of newer ARMulator

The latest ARMulator is supplied in two forms. You normally use the ARMulator supplied in file `armulate.dll`. If, however, you need to simulate the ARM966E-S core, then you must user the ARMulator supplied in file `armulxxe.dll` instead. Using either of these ARMulators supplied as part of the ARM Developer Suite, the ARMulation Configuration dialog appears as follows:



Variant: ARM9TDMI

Emulated [✓ Real-time] Speed:

Floating Point Emulation ☐

[✓ Little] Big

[✓ No Map File] Map File

Figure 9-12 Configuration of newer ARMulator

The configuration dialog for the newer ARMulator enables you to:

- specify which ARM processor you want ARMulator to emulate
- choose between emulating 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
- specify that the emulated target is to operate in little-endian or big-endian mode
- specify a memory map file, or that you want to use default settings.

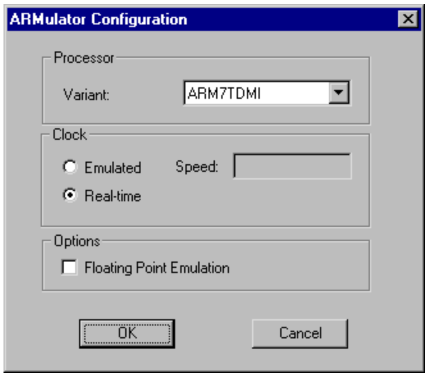
For information about ARMulator **Clock** speed settings, refer to *ARMulator clock speed* on page 9-34.

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 `armul.cnf` 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 the ARM Developer Suite, the ARMulation Configuration dialog appears as follows:



Variant: ARM7TDMI
Clock: Emulated ( ) [✓ Real-time] Speed:
Floating Point Emulation ☐

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 emulate
- choose between emulating 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 9-34.

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).

——— Note ———

If you use the SDT 2.50 `armulate.dll` file, you must use the SDT 2.50 `armul.cnf` file. If you use the ADS `armulate.dll` file, you must use the ADS `armul.cnf` file.

ARMulator clock speed

If you set a nonzero emulated **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 0.00 MHz 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 `armul.cnf` is used.

The configuration file is `armul9xxe.cnf` if you are using the special ARMulator from file `armul9xxe.dll`.

For ARMulator, an unspecified clock frequency is of no consequence because ARMulator does not need a clock frequency to be able to emulate 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 emulated 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.

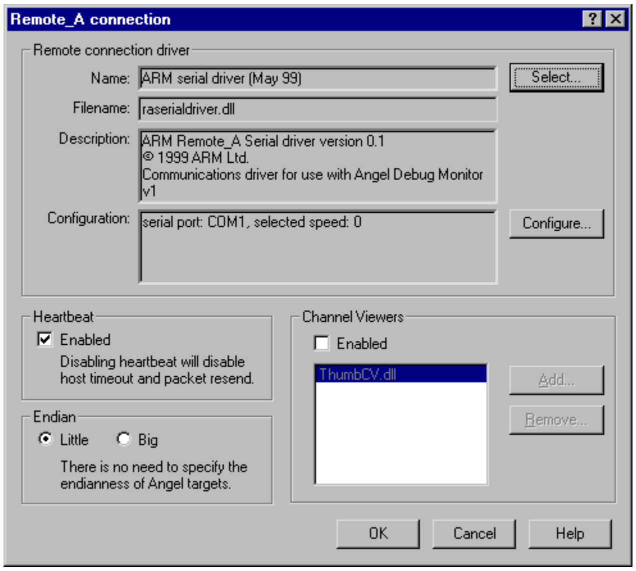
`$memstats` is handled slightly differently because it does need a defined clock frequency, so that ARMulator can calculate how many wait states are needed for the memory speed defined in an `armsd.map` file. If a clock speed is specified and an `armsd.map` file is present, then `$memstats` can give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

9.4.3 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.

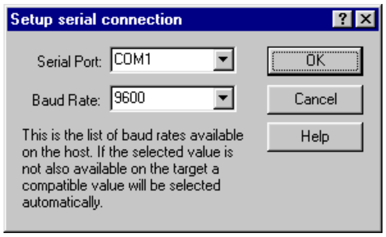
Channel Viewers
[ ] Enabled
ThumbCV.dll

Figure 9-14 Configuration of remote connection

The Remote_A connection dialog allows you to examine and/or 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 on page 9-36, or Figure 9-17 on page 9-36.

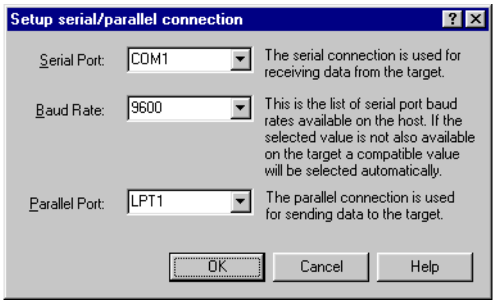


Serial Port: COM1

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



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

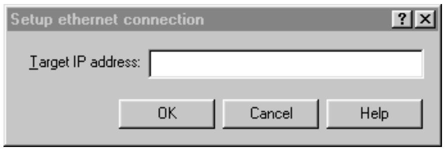


Figure 9-17 Ethernet connection configuration

Heartbeat

Ensures reliable transmission by sending heartbeat messages. If not enabled, there is a danger that the host and the target can get into a deadlock situation, with both waiting for a packet.

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, and 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. The only one ARM supplies is `ThumbCV.dll`. See *ThumbCV channel viewer* on page 9-23 for more information.

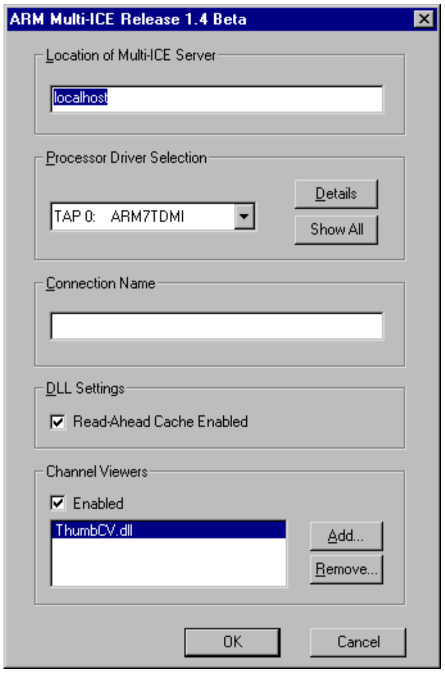
Click the **Add...** button to add a channel viewer DLL to the displayed list.

Click the **Remove...** button to remove the currently selected channel viewer DLL from the displayed list.

9.4.4 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 dialog shown in Figure 9-18.


ARM Multi-ICE Release 1.4 Beta

Location of Multi-ICE Server: localhost

Processor Driver Selection: TAP 0: ARM7TDMI

Connection Name:

DLL Settings: [✓] Read-Ahead Cache Enabled

Channel Viewers: [✓] Enabled

ThumbCV.dll

Figure 9-18 Multi-ICE configuration dialog

The Multi-ICE configuration dialog enables you to:

- specify the network address of the computer on which the Multi-ICE Server software is running
- select a processor driver
- specify a connection name (required only when access to the Multi-ICE Server software is across a network)
- specify DLL Settings to control the use of various debugger features

  Note

  The only such feature at present is Read Ahead Cache Enabled. This improves memory read performance by reading more memory than the debugger requests and caching the rest in case it is needed. The DLL learns which regions of memory are safe to access from previous read requests and never reads memory that has not been accessed previously. For certain operations this improves performance considerably, for example stepping with many string variables displayed in a debugger window.

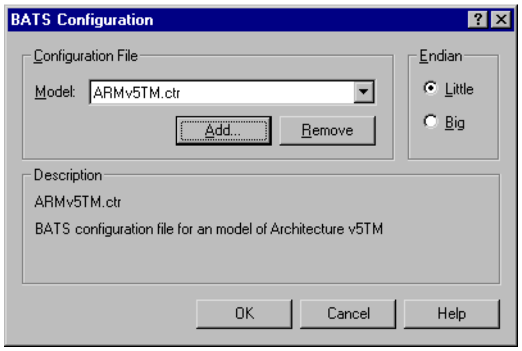
  The setting is saved and is on by default. If you are debugging a system with demand paged memory, switch this feature off.

- select a channel viewer (check the **Enabled** check box if you want to add viewers to or remove viewers from the list or to select one of the listed viewers).

9.4.5 BATS configuration

If you need to add BATS to the list of available targets, click **Add** and use the resulting browse dialog to locate and select the `bats.dll` file.

Select the BATS target line and click the **Configure** button to display the dialog shown in Figure 9-19.


BATS Configuration

Configuration File

Model: ARMv5TM.ctr

Endian: (● Little) ○ Big

Description

ARMv5TM.ctr

BATS configuration file for an model of Architecture v5TM

Figure 9-19 BATS configuration dialog

In the BATS configuration dialog you can:

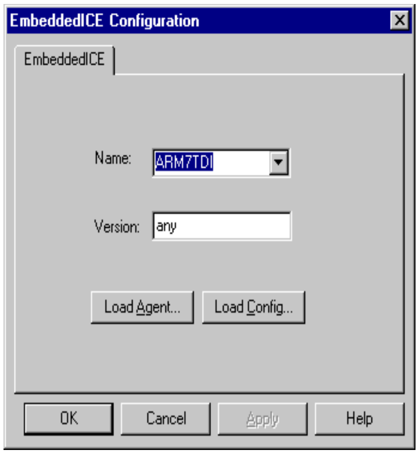
- click **Add** to display a browse dialog and select a filename to add to the (initially empty) list of available configuration files
- click **Remove** to remove the currently displayed filename from the list of available configuration files
- select little-endian or big-endian mode of operation for the target you are preparing to emulate
- click **OK** to configure BATS with the information stored in the configuration file identified by the currently displayed filename
- click **Cancel** to close the dialog without making any change to the current configuration of BATS.

9.4.6 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 9-20, is displayed.



Name: ARM7TDI
Version: any

Figure 9-20 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.

Chapter 10
Using ADW and ADU with C++

This chapter describes the additions that ARM C++ makes to ADW and ADU, and contains the following sections:

10.1 About ADW and ADU for C++

ADW and ADU have been extended to support C++ debugging. A dynamic link library (`adw_cpp.dll`) is installed in the same directory as `adw.exe`. The `adw_cpp.dll` adds:

- a **C++** menu between the **View** and **Execute** menus in the main menu bar
- five buttons in the ADW or ADU toolbar:

  Evaluate Expression

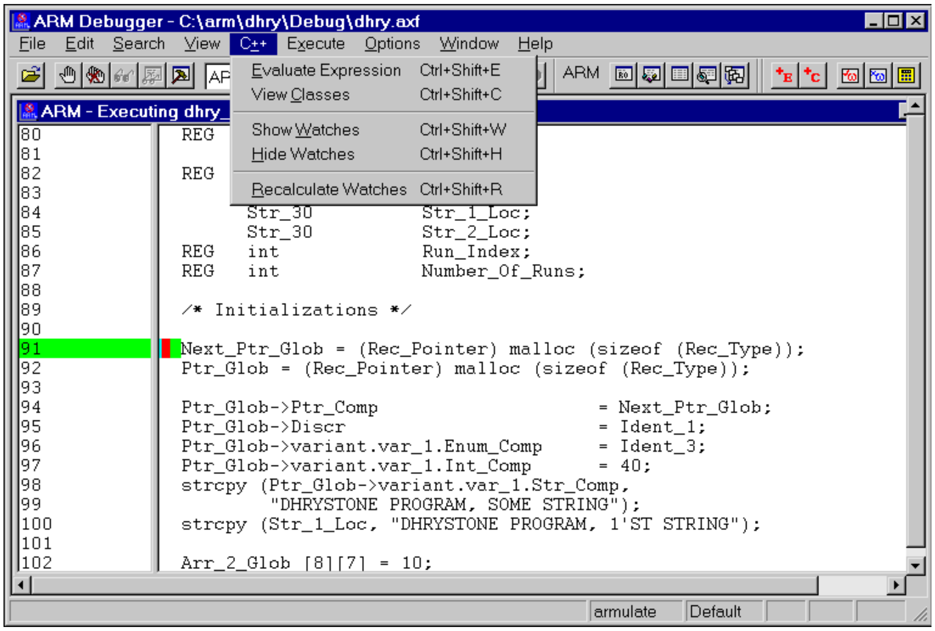
  View Classes

  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:

- the Class View window displays the class hierarchy of a C++ program in outline format
- the Watch View window displays a list of watches, allowing you to add and remove variables and expressions to be watched, and change the contents of watched variables
- the Evaluate Expression window 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.

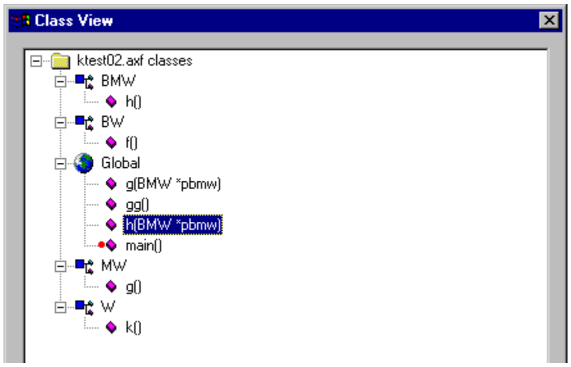


Figure 10-2 The Class View window

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 (Figure 10-3).

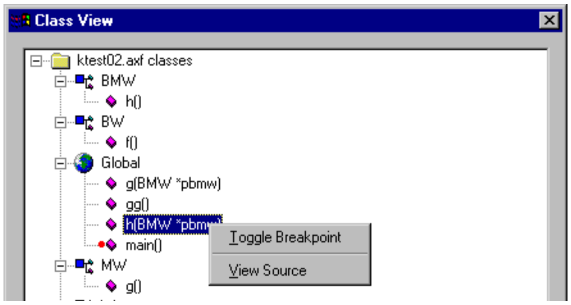


Figure 10-3 The Class View window menu

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* 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 Figure 10-3).
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 9-6 for more information.

Evaluation of function pointers and member functions is not available in this version of ADW or ADU.

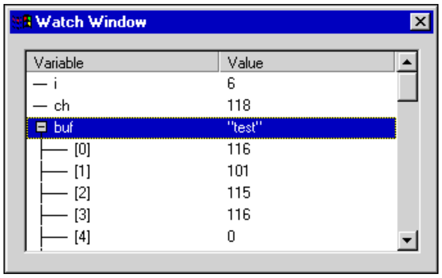
You can specify the contents and format of the Watch window using the Watch window menu. The following sections describe how to:

- 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.



Variable	Value
i	6
ch	118
buf	"test"
[0]	116
[1]	101
[2]	115
[3]	116
[4]	0

Figure 10-4 The 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 10-9 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 that it contains will 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`.

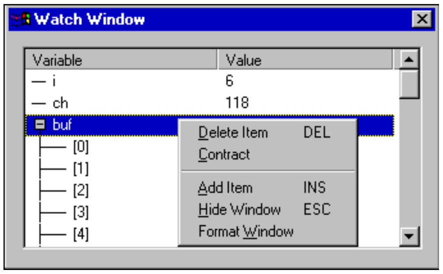


Figure 10-5 The Watch window menu

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).

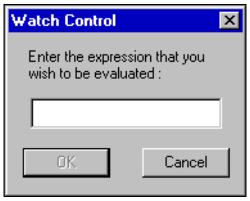


Figure 10-6 The Watch Control window

3. Enter an expression to add to the Watch window and click **OK**. Refer to *Evaluating expressions and adding watches* on page 10-9 for more information on the types of expression you can add to the Watch window.

Note

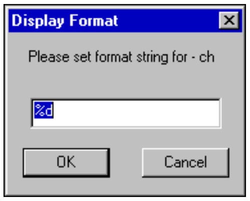
You can also add an expression to the Watch window directly from the Evaluate Expression window. Refer to *Evaluating expressions and adding watches* on page 10-9 for more information.

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

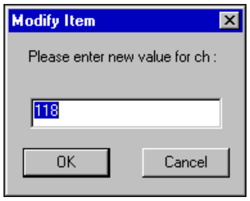
Figure 10-7 The Display Format window

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 9-12).
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.



Please enter new value for ch :

118

Figure 10-8 The 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. This menu item can be used if the value of any variable might have been changed by external hardware while the debugger is not stepping through code.

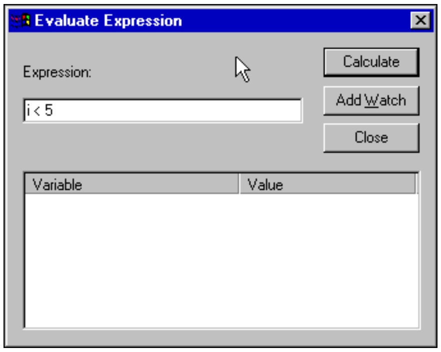
10.2.3 Evaluating expressions

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).



i < 5

Figure 10-9 The 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* 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++:

- Member functions of C++ classes cannot be used in expressions.
- Overloaded functions cannot be used in expressions.
- Only C operators can be used in constructing expressions. Any operators defined in a C++ class that also have a meaning in C, such as `[]`, will not work correctly because ADW and ADU use the C operator instead. Specific C++ operators, such as the scope operator `::`, are not recognized.
- Base classes cannot be accessed 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 will give undefined results.
- **`private`**, **`public`**, and **`protected`** attributes are not are not recognized in ADW or ADU expression evaluation. This means that private and protected member variables can be used during expression evaluation because ADW and ADU treat them as public.

Part C
armsd

Chapter 11
About armsd

The *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.

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, if those programs have been produced with debugging enabled. A limited amount of debugging information can be produced at link time, even if the object code being linked was not compiled with debugging enabled.

11.1.1 Selecting a debugger

armsd supports:

- remote debugging
- debugging using ARMulator
- debugging using BATS
- 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 such 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 `HOME`.

11.2 Command syntax

You invoke armsd using the command given below. Underlining is used to show the permitted abbreviations.

The full list of commands available when armsd is running is given in *Alphabetical list of armsd commands* on page 13-6.

11.2.1 Command-line options

```
armsd [-help] [-vsn] [-little|-big] [-proc name] [-fpe|-nofpe]
[-symbols] [-o name] [-script name] [-exec] [-iname] [-clock n]
[-remote|-armul|-bats|-adp options] image_name arguments
```

where:

`-help`	gives a summary of the armsd command-line options.
`-vsn`	displays information on the armsd version.
`-little`	specifies that memory should be little-endian (normally the default setting).
`-big`	specifies that memory should be big-endian.
`-proc` *name*	specifies the cpu type that is to be emulated. With this option you should not specify `-rem` or `-adp` as the target. Specify `-armul` as the target to invoke ARMulator or `-bats` to invoke BATS. If you do not specify a target, ARMulator is invoked if it can emulate the specified processor, BATS is invoked otherwise. If the specified processor cannot be emulated, `armsd` exits.
`-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`	asks the debugger to execute 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 only valid with an `armsd.map` file.
`-remote`	selects remote debugging. By default this will be ADP.
`-armul`	selects ARMulator (ARM emulator software). This is assumed by default if you do not specify a target but do specify a processor type that ARMulator can emulate.
`-bats`	selects the *Basic ARM Ten System* (BATS). This is assumed by default if you do not specify a target but do specify a processor type that ARMulator can not emulate.
`-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/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/parallel communications, you can prefix `,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 12-15 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 12-16 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 13-27 for more information.

`arguments` gives program arguments. You can also specify this information using the `load` command. See *load* on page 13-27 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 variables, program locations, expressions and constants.

12.1.1 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, variables defined in the current context can be accessed by giving their names. Other variables should be preceded by the context (for example, filename 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 in this case is:

```
procedure:variable
```

Global variables

Global variables can be referenced by qualifying them with the module name or filename if there is likely to be any ambiguity. For example, because the module name is the same as a procedure name, you should prefix the filename or module name with #. The syntax in this case 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 will look 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 needed to differentiate between different objects rarely need to be used together.

12.1.2 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, low-level locations can also be specified. 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 13-11) 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.

12.1.3 Expressions

Some debugger commands require expressions as arguments. Their syntax is based on C. A full set of operators is available. The lower the number, the higher the precedence of the operator. 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`

Table 12-1 Precedence of operators (continued)

Precedence	Operator	Purpose	Syntax
	`&`	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`
	`!=`	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`

Subscripting can only be applied to pointers and array names. The symbolic debugger checks both the number of subscripts and their bounds, in languages which support such checking. It is inadvisable to use out-of-bound array accesses. As in C, the name of an array can be used without subscripting to yield the address of the first element.

The prefix indirection operator `*` is used 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 non-primitive values.

12.1.4 Constants

Constants can be decimal integers, floating-point numbers, octal integers or hexadecimal integers. Note that `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.

Address constants can be specified by the address preceded with an `@` symbol. For commands which accept low-level symbols by default, the `@` can be omitted.

12.2 armsd variables

This section lists the variables available in armsd, and gives information on manipulating them.

12.2.1 Summary of armsd variables

Many debugger defaults can be modified 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
`$clock` (ARMulator only)	Number of microseconds since simulation started. This read-only variable is available only if a processor clock speed is specified. See *ARMulator configuration* on page 5-54 for information on specifying the emulated processor clock speed.
`$cmdline`	Argument string for the debuggee.
`$echo`	Non-zero if commands from obeyed files should be echoed (initially 1).
`$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.
`$memory_statistics` (ARMulator only)	Outputs any memory map statistics which ARMulator has been keeping. A read-only variable. See *ARMulator configuration* on page 5-54 for further details.
`$rdi_log`	RDI logging is enabled if non-zero, and serial line logging is enabled if bit 1 is set (initially 0).
`$result`	Integer result returned by last called function (junk if none, or if an integer result was not returned). A read-only variable.
`$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 should be caught when various conditions arise. The default value is `%RUsPDAifE`. Capital letters indicate that the condition 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)
`$type_lines`	Default number of lines for the `type` command.
`$top_of_memory`	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

`$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:
	`"c:\\mysource\\src1"`
	• separate multiple pathnames with a semicolon, not with a space character. For example:
	`"c:\\my src\\src1;c:\\my src\\src2"`

`$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: **0** specifies that no floating-point code is to be used (`none`) **1** selects software floating-point library with pure-endian doubles (`softVFP`), and is the default setting for images built with ADS tools **2** selects software floating-point library with mixed-endian doubles (`softFPA`) **3** selects hardware Vector Floating-Point unit (`VFP`) **4** selects hardware Floating-Point Accelerator (`FPA`). SoftVFP and SoftFPA images run correctly on a target whether or not hardware floating point is present, but VFP and FPA images must be run on the appropriate hardware.

armsd internal variables

The variables in Table 12-3 are included to support EmbeddedICE.

Table 12-3 armsd variables for 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.
`$semihosting_thumb_swi`	Defines which Thumb SWIs are interpreted as semihosting requests by the debug agent.

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}*
```

An equals sign(=) or a colon(:) can be used to separate the variable or location from the expression. If multiple expressions are used, they must be separated by commas or spaces.

Variables can only be changed to compatible types of expression. However, the debugger will convert 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 multiple expressions are specified, 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%.81x"` to another value with a command of the form:

```
{let} $int_format = string
```

The initial value of each of these format variables is given in *Summary of armsd variables* on page 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`). You cannot include high-level debugging tables in an image without also including low-level debugging tables.

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.

Low-level symbols can be used 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. Low-level symbols can also be used where a command would expect 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 `^`.

Memory addresses can also be used with commands and should 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 which is about to execute. This can include the condition code flags, interrupt enable flags, and processor mode bits, depending on the target ARM architecture. Note that this value can be different from the real value of register 15 due to the effect of pipelining.
`pc`	The address of the instruction which 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 current mode's program status register. The values displayed for the condition code flags, interrupt enable flags, and processor mode bits, are an alphabetic letter per condition code and interrupt enable flag, and a mode name (preceded by an underscore) for the mode bits. This mode name will be one of USER, IRQ, FIQ, SVC, UNDEF, ABORT and SYSTEM. 26-bit mode is no longer supported by the ARM tool chain. See also *Application Note 11, Differences Between ARM6 Series and Earlier Processors*.
`spsr`	`spsr` is the saved status register for the current mode. The values displayed are listed above in `psr` and `cpsr`. `spsr` is not defined if the processor is not capable of 32-bit operation.
`f0` to `f7`	The floating-point registers 0 to 7.
`fpsr`	The floating-point status register.
`fpcr`	The floating-point control register.
`a1` to `a4`	These refer to arguments 1 to 4 in a procedure call (stored in r0 to r3).
`v1` to `v7`	These refer to the five to seven general-purpose register variables which the compiler allocates (stored in r4 to r10).

Table 12-4 High-level symbols for low-level entities (continued)

Symbol	Description
`sb`	Static base, as used in reentrant variants of the *ARM/Thumb Procedure Call Standard* (ATPCS) (r9/v6).
`sl`	The stack limit register, used in variants of the APCS which implement software stack limit checking (r10/v7).
`ip`	Used in procedure entry and exit and as a scratch register (r12).

Printing register information

All these registers can be examined with the `print` command and changed 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, the `N` and `F` flags could be set, the `V` flag cleared, and the `I`, `Z` and `C` flags left untouched and the processor set to 32-bit supervisor mode, by typing:

```
let #psr = %NvF_SVC32
```

The following example changes to User mode:

```
psr = %_User32
```

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 need 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`.

12.4 armsd commands for EmbeddedICE

The following armsd commands are included for compatibility with EmbeddedICE. These are deprecated, and may be removed from future tool kits.

12.4.1 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.

12.4.2 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.

12.4.3 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 12-17.

Syntax

The syntax of the `loadconfig` command is:

```
loadconfig file
```

where:

`file`	names the EmbeddedICE configuration data file to load.

12.4.4 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 12-16).

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 which should be used:

`any`	accepts any version number. This is the default.
`n`	uses version *`n`*.
`n``+`	uses version *`n`* or later.

12.5 Accessing the debug communications channel

The debugger accesses the debug communications channel using the following commands.

For more information, see *Command-line debugging commands* on page A-3.

12.5.1 ccin

The `ccin` command selects a file containing Communications Channel data for reading. This command also enables Host to Target Communications Channel communication.

Syntax

The syntax of the `ccin` command is:

```
ccin filename
```

where:

`filename`	names the file containing the data for reading.

12.5.2 ccout

The `ccout` command selects a file where Communications Channel data is written, and also enables Target to Host Communications Channel communication.

Syntax

The syntax of the `ccout` command is:

```
ccout filename
```

where:

`filename`	names the file where the data is written.

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 13-6.

The functional groups are:

- *Symbols*
- *Controlling execution*
- *Reading and writing memory* on page 13-3
- *Program context* on page 13-3
- *Low-level debugging* on page 13-3
- *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 (variables) 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.
`return`	Returns to the caller of the current procedure (passing back a result).
`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 current context's caller.
`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 examine and 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 *program counter* (pc) and the status flags contained in the *program status register* (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's coprocessor support 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.

`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 the debugged program's variables.
`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.

13.2 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 13-8.

13.2.1 Annotating the command syntax

`typewriter`	Shows command elements that you should type at the keyboard.
<u>`type`</u>`writer`	Underlined text shows the permitted abbreviation of a command.
`typewriter`	Represents an item such as a filename or variable name. You should replace this with the name of your file, variable, and so on.
`{}`	Items in braces are optional. The braces are used for clarity and should not be typed.
`*`	A star (*) following a set of braces means that the items in those braces can be repeated as many times as required. Many command names can be abbreviated. The braces here show what can be left out. In the one case where braces are required by the debugger, these are enclosed in quote marks in the syntax pattern.

13.2.2 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 12-4. 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. High-level variables can be included 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. Low-level symbols can be included 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 12-3).

Variable A reference to one of the program's variables. 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 the 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

13.2.3 ! 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.

13.2.4 | 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.

13.2.5 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 no arguments are given, 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.

13.2.6 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.

13.2.7 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.

13.2.8 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. With no *size* specifier, `break` tries to determine the size of breakpoint to use by extracting information from the nearest symbol at or below the address to be broken. This usually chooses the correct size, if debug information is available. You can set the size explicitly, however, when setting breakpoints on ROM, for example.
loc	specifies where the breakpoint is to be inserted. For more information, see *Program locations* on page 12-3.
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 in the pattern above by braces within quotes. Each command 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 to display the line.
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. This is described in *unbreak* on page 13-39.

13.2.9 call

The `call` command calls a procedure.

Syntax

The syntax of the `call` command is:

```
call {/size} loc {(expression-list)}
```

where:

`/size` specifies which code type to break:

`/16`	specifies the instruction size as Thumb.
`/32`	specifies the instruction size as ARM.

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 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.

13.2.10 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 13-14.

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:

```
copro 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:	
		w*n*	The whole of the *n*th word of the register value
		w[*bit*]	Bit *bit* of the *n*th word of the register value
		w*n*[*bit1*:*bit2*]	Bits *bit1* to *bit2* inclusive of the *n*th word of the register value. The bits can be given in either order.
field '{' string {string} '}'*	*field*	One of the forms w*n*[*bit*] or w*n*[*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 w*n*[*bit*] or w*n*[*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.	

13.2.11 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 which take a context as an argument. When program execution is suspended, the search context is set to the active procedure.

13.2.12 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 13-14.

13.2.13 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 13-14.

13.2.14 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 13-13.

13.2.15 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*.

The `$examine_lines` variable can be used 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.

13.2.16 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.

13.2.17 fpregisters

The `fpregisters` command displays the contents of the eight floating-point registers f0 to f7 and the *Floating-point Program 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
```

(Note that `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	is a precision/format specifier:
	`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 indicates the floating-point mask and the second the floating-point flags. The status of the floating-point mask and flag bits is indicated by their case. Uppercase means the flag is set and lowercase means that it is cleared.

The flags are:

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

13.2.18 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, non-zero), 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.

13.2.19 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.

13.2.20 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 *`.

13.2.21 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.

13.2.22 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.

13.2.23 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.

13.2.24 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

The `let` command is used 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 multiple expressions are used, they must be separated 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 multiple expressions are specified, each expression is assigned to `variable[`*n*`-1]`, where *n* is the nth expression.

See also *let* on page 12-10 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;home/test source/lib1"
ARMSD: p $sourcedir
"/home/usr/me/src;/home/usr/me/src2;home/test source/lib1"
```

Note

No warning is displayed if you enter an invalid pathname.

Command-line arguments

Command-line arguments for the debuggee can be specified using the `let` command with the root-level variable `$cmdline`. The syntax in this case is:

{let} $cmdline = *string*

The program name is automatically passed as the first argument, and thus should not be included in the string. The setting of `$cmdline` can be examined using `print`.

`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 = value
```

```
print /%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 such a case, enter `LL` or `ULL` as appropriate at the end of the new value to force its acceptance.

13.2.25 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.

13.2.26 load

The `load` command loads an image for debugging.

Syntax

The syntax of the `load` command is:

<u>`lo`</u>`ad{/`*`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

`image-file` and any necessary *`arguments`* can also be specified 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.

13.2.27 log

The `log` command sends the output of subsequent commands to a file as well as 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 terminate logging, type `log` without an argument. The file can then be examined using a text editor or the `type` command.

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 at the start and/or end of the pattern:

`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 which 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 be that of the caller of the current context.

Syntax

The syntax of the `out` command is:

```
out
```

Usage

It is an error to issue an `out` command when no further movement in that direction is possible.

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 13-7. 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`.
expression	enters the expression for evaluation.

Usage

Pointer values are treated as integers, using a default fixed format `%.8x`, for example, `000100e4`.

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 Tools 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 Tools 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 Tools 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.

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 the 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, via 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 status flags contained in the program status register.

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 = 0x00008000     PSR= %NzcVIF_SVC26
```

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:

`reload{/profile-option} {arguments}`

where

`profile-option`	specifies which profiling option to use:	
	`/callgraph`	tells the debugger to provide the image being loaded with counts to enable the dynamic call-graph profile to be constructed.
	`/profile`	directs the debugger to prepare the image being loaded for flat profiling.
`arguments`	are the command-line arguments the program normally takes. If no *`arguments`* are specified, the arguments used in the most recent load or reload setting of `$cmdline` or command-line invocation are used again.	

Usage

Breakpoints (but not watchpoints) remain set after a `reload` command.

13.2.42 return

The `return` command returns to the caller of the current procedure, passing back a result where required.

Syntax

The syntax of the `return` command is:

`return {expression}`

where:

`expression`	contains the expression to be evaluated.

Usage

You cannot specify the return of a literal compound data type such as an array or record using this command, but you can return the value of a variable, expression or compound type.

13.2.43 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 which immediately follows that function.
`count`	specifies the number of statements to be stepped through: if it is omitted only one statement will be 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`.

13.2.44 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:

<u>`sy`</u>`mbols {`*`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`

13.2.45 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.

13.2.46 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.

13.2.47 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.

13.2.48 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

`variable` can also return the type of an expression.

13.2.49 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 *`variable`* is not specified, 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/EmbeddedICE logic, ensure that watchpoints use hardware watchpoint registers to avoid any performance penalty.

Note

When using the C compiler, be aware that the code produced can use the same register to hold more than one variable if their lifetimes do not overlap. If the register variable you are investigating is no longer being used by the compiler, you might see a value pertaining to a completely different variable.

Adding watchpoints can make programs execute very slowly, because the value of variables has to be checked every time they could have been altered. It is more practical to set a breakpoint in the area of suspicion and set watchpoints once execution has stopped.

13.2.50 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.

13.2.51 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).

Appendix A
Debug Communications Channel

This appendix explains the use of the Debug Communications Channel. It contains the following sections:

A.1 Introduction

The EmbeddedICE logic in ARM cores such as ARM7TDMI and ARM9TDMI contains a debug communication channel. This allows data to be passed between the target and the host debugger using the JTAG port and a protocol converter such as Multi-ICE, without stopping the program flow or entering debug state. This appendix examines how the debug communication channel can be accessed by a program running on the target and by the host debugger.

——— **Note** ———

The EmbeddedICE logic in ARM7DI does not implement a debug communications channel.

If the ARM-based system you are using makes use of an AMBA rev. C ARM7TDMI wrapper, then the debug communicatiuons channel will not work. Current (Sept. 98) versions of the Atmel AT91 suffer from this problem.

———

ADS provides three methods of accessing the debug communication channel:

- a command-line debugger, such as:
  - armsd
  - the command window in ADW
- the Channel Viewer mechanism in AXD or ADW
- Multi-ICE semihosting.

——— **Note** ———

If you wish to make use of the facilities described in this appendix with an EmbeddedICE interface, ensure that you are using EmbeddedICE agent software version 2.04 or later (2.07 is the latest version at the time of writing), and GAL version EFI-0011C.

———

For further information on the debug facilities provided by EmbeddedICE on the ARM7TDMI, see:

- the technical reference manual or datasheet for the ARM core that you are using
- other documentation supplied with ADS, as listed in the preface to this book.

A.2 Command-line debugging commands

To access the debug communication channel from a command line, using armsd or the command-line window in ADW, use the following commands:

`ccin` *`filename`*

Selects a file containing comms channel data for reading. This command also enables host to target comms channel communication.

`ccout` *`filename`*

Selects a file where comms channel data is written. This command also enables target to host comms channel communication.

A.3 Enabling comms channel viewing

Debug communications channel viewing is supported in both AXD and ADW.

A.3.1 Comms channel viewing in AXD

To enable channel viewing in AXD, refer to *Control system view pop-up menus* on page 5-34.

To use a channel viewer in AXD, refer to *Comms Channel processor view* on page 5-25.

A.3.2 Comms channel viewing in ADW

To enable channel viewing and to use a channel viewer in ADW, refer to *Channel viewers* on page 9-23.

A.4 Target transfer of data

The debug communication channel is accessed by the target as coprocessor 14 on the ARM core using the ARM instructions `MCR` and `MRC`.

Two registers are provided to transfer data:

Comms data read register

A 32-bit wide register used to receive data from the debugger. The following instruction returns the read register value in `Rd`:

```
MRC p14, 0, Rd, c1, c0
```

Comms data write register

A 32-bit wide register used to send data to the debugger. The following instruction writes the value in `Rn` to the write register:

```
MCR p14, 0, Rn, c1, c0
```

A.5 Polled debug communications

In addition to the comms data read and write registers, a comms data control register is provided by the debug communication channel.

The following instruction returns the control register value in Rd:

```
MRC p14, 0, Rd, c0, c0
```

Two bits in this control register provide synchronized handshaking between the target and the host debugger:

Bit 1 (W bit)

Denotes whether the comms data write register is free (from the target's point of view):

W = 0	New data may be written by the target application.
W = 1	The host debugger can scan new data out of the write register.

Bit 0 (R bit)

Denotes whether there is new data in the comms data read register (from the target's point of view):

R = 1	New data is available to be read by the target application.
R = 0	The host debugger can scan new data into the read register.

Note

The debugger cannot use coprocessor 14 to access the debug communication channel directly, as this has no meaning to the debugger. Instead, the debugger can read from and write to the debug communication channel registers using the scan chain. The debug communication channel data and control registers are mapped into addresses in the EmbeddedICE logic.

A.5.1 Target to debugger communication

This is the sequence of events for an application running on the ARM core to communicate with the debugger running on the host:

1. The target application checks if the debug communication channel write register is free for use. It does this using the MRC instruction to read the debug communication channel control register to check that the W bit is clear.

2. If the W bit is clear, the debug communication write register is clear and the application writes a word to it using the MCR instruction to coprocessor 14. The action of writing to the register automatically sets the W bit. If the W bit is set, the debug communication write register has not been emptied by the debugger. If the application needs to send another word, it must poll the W bit until it is clear.

3. The debugger polls the debug communication control register via scan chain 2. If the debugger sees that the W bit is set, it can read the debug communication channel data register to read the message sent by the application. The process of reading the data automatically clears the W bit in the debug communication control register.

   The following piece of target application code, supplied in file `Examples/dcc/outchan.s`, shows this in action:

```
     AREA  OutChannel, CODE, READONLY
     ENTRY
     MOV   r1,#4           ; Number of words to send
     ADR   r2, outdata     ; Address of data to send
pollout
     MRC   p14,0,r0,c0,c0 ; Read control register
     TST   r0, #2
     BNE   pollout         ; if W set, register
                           ; still full
write
     LDR   r3,[r2],#4      ; Read word from outdata
                           ; into r3 and update the
                           ; pointer
     MCR   p14,0,r3,c1,c0 ; Write word from r3
     SUBS  r1,r1,#1        ; Update counter
     BNE   pollout         ; Loop if more words to
                           ; be written
     MOV   r0, #0x18       ; Angel_SWIreason_ReportException
     LDR   r1, =0x20026    ; ADP_Stopped_ApplicationExit
     SWI   0x123456        ; ARM semihosting SWI
outdata
     DCB "Hello there!"
     END
```

4. Assemble and link this code using the following commands:

```
armasm -g outchan.s
armlink outchan.o -o outchan.axf
```

You have created an executable image in a file called `outchan`. Your next steps depend on your choice of debugger. You can load the image, enable comms channel viewing, and execute the image by using:

- armsd

- AXD
- ADW.

Using armsd

If you are using armsd:

1. Load the image into armsd with the command:

   ```
   armsd -li -adp -port s=1 outchan.axf
   ```

2. Enable communication and open the output file, then execute the program:

   ```
   ccout output
   go
   ```

3. Quit armsd when execution finishes. You should be able to view the file and see that transfer has occurred.

Using AXD

If you are using AXD:

1. Enable channel viewing, as described in *Control system view pop-up menus* on page 5-34.
2. Load the image created above into AXD.
3. Use the channel viewer in AXD, as described in *Comms Channel processor view* on page 5-25.
4. In the AXD main screen, select **Go** from the **Execute** menu (or press **F5**) to execute the image.

   The data sent from the target (in this example, `Hello there!`) should now be displayed in the Channel Viewer window.

Using ADW

If you are using ADW:

1. Enable channel viewing and load the image, as described in *Channel viewers* on page 9-23.
2. In the **Channel Viewer** window, select **Start Viewer** from the **Control** menu.
3. In the **ADW** main window select **Go** from the **Execute** menu, to execute the program.

The data sent from the target (in this example, `Hello there!`) should now be displayed in the Channel Viewer window.

A.5.2 Debugger to target communication

This is the sequence of events for message transfer from the debugger running on the host to the application running on the core:

1. The debugger polls the debug communication control register `R` bit. If the `R` bit is clear, the debug communication read register is clear and data can be written there for the target application to read.

2. The debugger scans the data into the debug communication read register via scan chain 2. The `R` bit in the debug communication control register is automatically set by this.

3. The target application polls the `R` bit in the debug communication control register. If it is set, there is data in the debug communication read register that can be read by the application, using the `MRC` instruction to read from coprocessor 14. The `R` bit is cleared as part of the read instruction.

   The following piece of target application code, supplied in file `Examples/dcc/inchan.s`, shows this in action:

```
     AREA  InChannel, CODE, READONLY
     ENTRY
     MOV   r1,#4           ; Number of words to read
     LDR   r2, =indata     ; Address to store data read
pollin
     MRC   p14,0,r0,c0,c0 ; Read control register
     TST   r0, #1
     BEQ   pollin          ; If R bit clear then loop
read
     MRC   p14,0,r3,c1,c0 ; read word into r3
     STR   r3,[r2],#4      ; Store to memory and
                           ; update pointer
     SUBS  r1,r1,#1        ; Update counter
     BNE   pollin          ; Loop if more words to read
     MOV   r0, #0x18       ; Angel_SWIreason_ReportException
     LDR   r1, =0x20026    ; ADP_Stopped_ApplicationExit
     SWI   0x123456        ; ARM semihosting SWI

     AREA  Storage, DATA, READWRITE
indata
     DCB   "Duffmessage#"
     END
```

4. Create an input file on the host containing, for example, `And goodbye!`.

5. Assemble and link this code using the following commands:

   ```
   armasm -g inchan.s
   armlink inchan.o -o inchan.axf
   ```

You have created an executable image in a file called `inchan`. Your next steps depend on your choice of debugger.

You can load the image, enable comms channel viewing, and execute the image by using:

- armsd (for command-line operation)
- AXD (the latest ARM debugger)
- ADW (an earlier ARM debugger).

Issuing commands

If you are issuing commands:

1. Load the image into armsd using the following command:

   ```
   armsd -li -adp -port s=1 inchan.axf
   ```

   If you view the area of memory indata, you see its initial random contents:

   ```
   examine indata
   ```

2. Enable communication and open the input file, then execute the program:

   ```
   ccin input
   go
   ```

3. When execution completes, view memory again and you can see the input has been read in:

   ```
   examine indata
   ```

Using AXD

If you are using AXD:

1. Enable channel viewing, as described in *Control system view pop-up menus* on page 5-34.
2. Load the image created above into AXD.
3. Use the channel viewer in AXD, as described in *Comms Channel processor view* on page 5-25.

4. In the **Send** field of the Channel Viewer, type `And goodbye!`, and click the **Send** button. The **Left to Send** counter should show the number of bytes stored for sending to the target.

   If you view the area of memory `indata`, you see its initial contents:

   ```
   examine indata
   ```

5. In the AXD main screen, select **Go** from the **Execute** menu (or press **F5**) to execute the image.

6. When execution is complete, view memory again and you can see that the input has been read in:

   ```
   examine indata
   ```

Using ADW

If you are using ADW:

1. Enable channel viewing and load the image, as described in *Channel viewers* on page 9-23.

2. In the **Channel Viewer** window, select **Start Viewer** from the **Control** menu.

3. In the **Edit** box on the dialog bar of the Channel Viewer, type `And goodbye!`, and click the **Send** button. The **Left to Send** counter should show the number of bytes stored for sending to the target.

   If you view the area of memory `indata`, you see its initial contents:

   ```
   examine indata
   ```

4. In the **ADW** main window select **Go** from the **Execute** menu, to execute the program.

5. When execution is complete, view memory again and you can see that the input has been read in:

   ```
   examine indata
   ```

A.6 Interrupt-driven debug communications

The examples given above are polled. It is also possible to convert these to interrupt-driven examples by connecting up `COMMRX` and `COMMTX` signals from the ARM7TDMI core to your interrupt controller.

The read and write code given above could then be moved into an interrupt handler.

For information on writing interrupt handlers refer to the *ADS Developer Guide*.

A.7 Access from Thumb state

As the Thumb instruction set does not contain coprocessor instructions, you cannot use the debug communication channel while the core is in Thumb state.

There are three possible ways around this:

- You can write each polling routine as a SWI (Software Interrupt), which can then be executed while in either ARM or Thumb state. Entering the SWI handler immediately puts the core into ARM state where the coprocessor instructions are available. Refer to the *ADS Developer Guide* for further information on SWIs.
- Thumb code can make interworking calls to ARM subroutines which implement the polling. Refer to the *ADS Developer Guide* for further information on mixing ARM and Thumb code.
- Use interrupt-driven communication rather than polled communication. The interrupt handler would be written in ARM instructions, so the coprocessor instructions can be accessed directly.

A.8 Semihosting

You can use the debug communications channel for semihosting if you are using Multi-ICE. For further information refer to the *Multi-ICE User Guide*.

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 program 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.
AOF	See *ARM Object Format*.

ARM Debugger for UNIX	*ARM Debugger for UNIX* (ADU) and *ARM Debugger for Windows* (ADW) are two versions of the same ARM debugger software, running under UNIX or Windows respectively. This debugger was issued originally as part of the *ARM Software Development Toolkit*. It is still fully supported and is now supplied as part of the *ARM Developer Suite*.
ARM Debugger for Windows	*ARM Debugger for Windows* (ADW) and *ARM Debugger for UNIX* (ADU) are two versions of the same ARM debugger software, running under Windows or UNIX respectively. This debugger was issued originally as part of the *ARM Software Development Toolkit*. It is still fully supported and is now supplied as part 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.
ARMulator	ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
ARM Object Format	A (now obsolete) format for object files.
armsd	The *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.
ATPCS	ARM/Thumb Procedure Call Standard.
AXD	See *ARM eXtended Debugger*.
Backtracing	See *Stack backtracing* and *Tracing*.
Basic ARM Ten System	The *Basic ARM Ten System* (BATS) is a modelling scheme similar to but separate from *ARMulator*. BATS is designed specifically to model systems based on the ARM10 processor. *ARMulator* models systems based on all earlier ARM processors.
BATS	See *Basic ARM Ten System*.
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*.
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.
EIDE	See *Integrated Drive Electronics*.
ELF	Executable Linkable Format.

Enhanced Program Status Register	See *Program Status Register*.
EPSR	Enhanced Program Status Register. See *Program Status Register*.
Executable image	See *Image*.
Execution Unit	A debugger object representing a thread of execution within an 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.
ICE	In-circuit Emulator.
IDE	See *Integrated Development Environment*.
Image	An 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.
JTAG	Joint Test Access Group. Many debug and programming tools use a 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).
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 in-circuit emulator. 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.
PIE	A platform-independent evaluator card 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 emulated 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. Often, therefore, also referred to as *Processor Status Register*. 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*.

Register A processor register.

RDI 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).

Remote_A A communications protocol used, for example, between debugger software such as *ARM eXtended Debugger* (AXD) and a debug agent such as *Angel*.

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 *Software Development Toolkit* (SDT) is an ARM product still supported but superseded by *ARM Developer Suite* (ADS).

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.

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.
TC	See *Transaction Control*.
Thread	A thread of execution on a processor.
	A context of execution on a processor. A thread is always related to a processor and may or may not be associated with an image.
Thumb state	A processor that is executing Thumb (16-bit) instructions is operating in Thumb 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*.
Variable	A named memory location of an appropriate size to hold a specific data item.
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.