

# ARM Debugger for UNIX

## User Guide

Copyright © 1998 ARM Limited. All rights reserved.

### **Proprietary Notice**

ARM™ and the ARM Powered™ logo are trademarks of ARM Limited.

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

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

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

# Preface

This preface introduces the ARM Debugger for UNIX (ADU) and its documentation.

The ARM Debugger for UNIX is an extra-cost optional addition to the ARM Software Development Toolkit (SDT). It requires SDT 2.11a or greater. This suite, together with supporting documentation and examples, enables you to write and debug applications for the ARM family of RISC processors.

Several references are made to C++ code in this guide. The ARM C++ compiler is not included in the ARM Software Development Toolkit, but may be purchased separately.

## About this book

After the table of contents, this book is organized into the following chapters:

**Chapter 1:** *Introduction*

A basic introduction to ADU.

**Chapter 2:** *Debugger Concepts*

An overview of the terminology used in ARM debugging.

**Chapter 3:** *Getting Started*

A step by step guide to debugging a simple application, and ways of displaying information while debugging.

**Chapter 4:** *Further Debugging Features*

An overview of the more advanced features of the debugger.

**Chapter 5:** *Configurations*

A description of the configuration options that allow you to control various settings used by the debugger, ARMulator, the Angel remote connection, and EmbeddedICE.

**Appendix A:** *FlexLM License Manager*

How to operate the debugger under the control of the FlexLM License Manager.

**Index**

A comprehensive alphabetically ordered index.

## Related publications

The *ARM Software Development Toolkit User Guide* (ARM DUI 0040) describes all the applications which together form the ARM Software Development Toolkit, version 2.11.

The *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) provides reference information on each of the ARM software tools, procedure call standards, file formats, and so on.

*Target Development System User Guide* (ARM DUI 0061) describes how you set up your target development system.

*C++ User and Reference Guide* (ARM DUI 0047) provides C++ specific information.

## 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 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* Introduces special terminology, denotes internal cross-references, and citations.

**bold** Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate.

`typewriter` **bold** Denotes language keywords when used outside example code.

## Feedback

You may wish to make comments or suggestions relating to either the documentation or the product.

### Feedback on this book

If you have any comments on this book, please contact your supplier giving:

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

General suggestions for additions and improvements are also welcome.

### Feedback on the ARM Debugger for UNIX

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

- details of the release you are using, in particular the version string of the tool, including the version number and date
- 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.

# Contents

## ARM Debugger for UNIX User Guide

	<b>Preface</b>	
	About this book .....	iv
	Feedback .....	vi
<b>Chapter 1</b>	<b>Introduction</b>	
	1.1 Online help .....	1-2
	1.2 Debugging an ARM application .....	1-3
	1.3 Debugging systems .....	1-4
<b>Chapter 2</b>	<b>Debugger Concepts</b>	
	2.1 Debugger concepts .....	2-2
	2.2 Debugger main operations .....	2-3
	2.3 Debugger further details .....	2-5
<b>Chapter 3</b>	<b>Getting Started</b>	
	3.1 The ADU desktop .....	3-2
	3.2 Using the debugger .....	3-14
	3.3 Displaying image information .....	3-23
<b>Chapter 4</b>	<b>Further Debugging Features</b>	
	4.1 Setting and editing complex breakpoints and watchpoints .....	4-2
	4.2 Saving or changing an area of memory .....	4-6
	4.3 Specifying command-line arguments for your program .....	4-8
	4.4 Using command-line debugger instructions .....	4-9
	4.5 Profiling .....	4-10
	4.6 Flash download .....	4-11
<b>Chapter 5</b>	<b>Configurations</b>	
	5.1 Debugger configuration .....	5-2
	5.2 ARMulator configuration .....	5-9
	5.3 Remote target configuration .....	5-12
	5.4 EmbeddedICE configuration .....	5-14

**Appendix A**

**FlexLM License Manager**

A.1 About license management ..... A-2

A.2 Obtaining your license file ..... A-4

A.3 What to do with your license file ..... A-5

A.4 Starting the server software..... A-6

A.5 Running your licensed software ..... A-7

A.6 Customizing your license file ..... A-9

A.7 Finding a license..... A-11

A.8 Using FlexLM with more than one product ..... A-12

A.9 FlexLM license management utilities..... A-14

A.10 Frequently asked questions about licensing..... A-18



# Chapter 1

## Introduction

The ARM Debugger for UNIX (ADU) enables you to debug your ARM targeted image using any of the debugging systems described in *Debugging systems* on page 1-4.

Refer to the documentation supplied with your target board for specific information on setting up your system to work with the ARM Software Development Toolkit, and the EmbeddedICE interface, Angel, and so on.

See Chapter 5 of the *ARM Software Development Toolkit User Guide* for more information on the ARMulator.

For detailed instructions on how to use ADU, refer to the comprehensive online help.

This chapter contains the following sections:

- *Online help* on page 1-2
- *Debugging an ARM application* on page 1-3
- *Debugging systems* on page 1-4.

## 1.1 Online help

When you have started ADU, you can use online help to find information on the tasks you are performing. You have several options for accessing the Help system:

- |                 |   |
|-----------------|---|
| <b>Contents</b> | Select <b>Contents</b> from the <b>Help</b> menu to display a Table of Contents.  |
| <b>Search</b>   | Select <b>Index</b> from the <b>Help</b> menu to display an index of all the help topics.                               |
| <b>Help</b>     | Click the <b>Help</b> button to get information on the dialog currently on display.                                     |
| <b>F1</b>       | Press the <b>F1</b> key on your keyboard to get help on the currently active window or the dialog currently on display. |

## 1.2 Debugging an ARM application

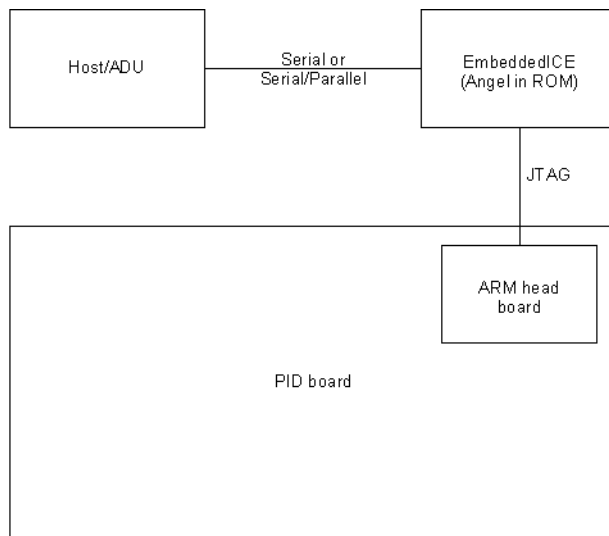
ADU works in conjunction with either a hardware or a software target system. An ARM Development Board, communicating through an EmbeddedICE interface or Angel, is an example of a hardware target system. ARMulator is an example of a software target system. You debug your application using a number of windows that give you various views on the application you are debugging.

There are a number of ways in which you can debug an application developed to run on an ARM-based system. This section helps you to choose a debugging strategy suitable for your application.

To debug your application you must choose:

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

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



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

## 1.3 Debugging systems

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

- the ARMulator
- EmbeddedICE interface
- Angel Debug Monitor.

The three systems listed above are described in the following sections.

### 1.3.1 The ARMulator

The ARMulator is a collection of programs that emulate the instruction sets and architecture of various ARM processors. It 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, the ARMulator is well suited to software development and benchmarking. The ARMulator also supports a full ANSI C library to allow complete C programs to run on the emulated system.

The ARMulator:

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

See Chapter 5 of the *ARM Software Development Toolkit User Guide* for more information.

### 1.3.2 EmbeddedICE interface

EmbeddedICE is a JTAG based debugging system for ARM processors. EmbeddedICE provides the interface between a debugger and an ARM core embedded within any ASIC. EmbeddedICE provides you with:

- real time address 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).

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

See Chapter 7 of the *ARM Software Development Toolkit User Guide* for information about using an EmbeddedICE interface and *EmbeddedICE configuration* on page 5-14 for information on configuration options.

### 1.3.3 Angel

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. The Angel Debug Monitor runs alongside the application being debugged on the target platform. There are two versions of Angel:

- a full version for use on development hardware
- a minimal version intended for use on production hardware.

You can use Angel to:

- evaluate existing application software on real hardware, as opposed to hardware emulation
- develop software applications on development hardware
- bring into operation new hardware that includes an ARM processor
- port operating systems to ARM-based hardware.

You can use Angel to debug an application on an ARM Development Board or on your own custom hardware. See Chapter 6 of the *ARM Software Development Toolkit User Guide* for more information.

### 1.3.4 Remote\_D

Remote\_D connections are not supported by the ARM Debugger for UNIX.

---

#### Note

Remote\_D has been superseded by the Angel Debug Monitor (Remote\_A). See Chapter 6 of the *ARM Software Development Toolkit User Guide* for more information.

---



# Chapter 2

## Debugger Concepts

This chapter introduces the terminology used in this guide, and includes brief explanations of the main concepts involved. It contains the following sections:

- *Debugger concepts* on page 2-2
- *Debugger main operations* on page 2-3
- *Debugger further details* on page 2-5.

## 2.1 Debugger concepts

This section introduces some of the concepts of which you need to be aware when debugging program images.

### 2.1.1 ARM/Thumb code

The ARM assembly language is 32-bit assembly code that can be assembled to run on all ARM cores.

The Thumb assembly language is 16-bit assembly code that can be assembled to run on a Thumb-capable ARM core, such as ARM7TDMI and ARM9TDMI.

### 2.1.2 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 ADU itself. Examples of debug agents include the EmbeddedICE interface and the Angel Debug Monitor.

### 2.1.3 Remote debugging interface

The Remote Debug Interface (RDI) is a procedural interface between a debugger and the image being debugged, through a debug monitor or controlling debug agent. RDI gives the debugger core a uniform way to communicate with:

- a controlling debug agent or debug monitor linked with the debugger
- a debug agent executing in a separate operating system process
- a debug monitor running on ARM-based hardware accessed through a communication link
- a debug agent controlling an ARM processor through hardware debug support.

See the *ARM Software Development Toolkit Reference Guide* for more information on RDI.



## 2.2 Debugger main operations

This section introduces breakpoints, watchpoints, backtrace, and disassembly. These are the debugger features that you use most often.

### 2.2.1 Breakpoints

A breakpoint is a point in the code where your program is halted by ADU. When you have set a breakpoint it appears as a red marker in the left-hand 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, use either the interleaved source option or the disassembly view.

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:

- Backtrace
- Breakpoints
- Disassembly
- Execution
- Function names
- Low-level symbols
- Source File.

### 2.2.2 Watchpoints

In its simplest form, a watchpoint halts a program when a specified register or variable is changed. 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 specified variable changes
- a complex watchpoint that:
  - stops when the variable has changed a specified number of times
  - stops when the variable is set 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.

---

### 2.2.3 Backtrace

When your program has halted, typically at a breakpoint or watchpoint, backtrace information is displayed in the Backtrace Window to give you 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:

```
#DHR_2:Proc_6 line 42
#DHR_1:Proc_1 line 315
#DHR_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**

---

An assembly language program assembled without debug information would show only the pc values.

---

### 2.2.4 Disassembled code

Disassembled code is the assembler code generated by the disassembly process.

You can display disassembled code in the Execution Window or in the Disassembly Window (select **Disassembly** from the **View** menu). You can also choose the type of disassembled code to display by accessing the **Disassembly mode** submenu, from the **Options** menu. ARM code, Thumb code, or both can be displayed, depending on the image type of your program.

For more information see *Disassembled code* on page 3-27.

## 2.3 Debugger further details

This section introduces other debugging features that you can use in conjunction with the main features already described.

Search paths, regular expressions, high-level and low-level symbols, and profiling are covered here.

### 2.3.1 Search paths

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 are using the ARM command-line tools to build your project, you may 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 ADU (see *Search paths* on page 3-23).

### 2.3.2 Regular expressions

Regular expressions are the means by which you 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.

Pattern matching is done following the UNIX `regex(5)` format, but without the special symbols, `^` and `$`.

The following special characters modify the meaning of the previous regular expression, and work only if such regular expression is given.

- |   |  |
|---|--|
| * | Zero or more of the preceding regular expressions. For example, <code>A*B</code> would match <code>B</code> , <code>AB</code> , and <code>AAB</code> .     |
| ? | Zero or one of the preceding regular expression. For example, <code>AC?B</code> matches <code>AB</code> and <code>ACB</code> but not <code>ACCB</code> .   |
| + | One or more of the preceding regular expression. For example, <code>AC+B</code> matches <code>ACB</code> and <code>ACCB</code> , but not <code>AB</code> . |

The following special characters are regular expressions in themselves:

- `\`                Precedes any special character that you want 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 may 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`.

### 2.3.3 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 may 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 Procedure Call Standard (APCS), or argument lists in other procedures. See also Chapter 10 of the *ARM Software Development Toolkit User Guide*. 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.

### 2.3.4 Profiling

Profiling allows you to see in which parts of your code most processor time is spent, by sampling the pc at specified time intervals. This information is used to build up a picture of the percentage time spent in each procedure. Using the command-line program `armprof` on the data generated by either `armsd` or `ADU`, you can see where effort can be most effectively spent to make the program more efficient.

---

**Note**

---

You cannot display profiling information from within `ADU`. You must capture the data using the **Profiling** functions on the **Options** menu, then use the `armprof` command-line tool.

Profiling support is provided by `ARMulator` and `Angel`, but not by the `EmbeddedICE` interface.

---

See *Profiling* on page 4-10 of this guide and Chapter 8 of the *ARM Software Development Toolkit User Guide* for more information on profiling.



# Chapter 3

## Getting Started

This chapter describes the ARM Debugger for UNIX desktop, including details of the windows that are available during your debugging session. It also provides step-by-step instructions explaining how to perform typical debugging tasks.

It contains the following sections:

- *The ADU desktop* on page 3-2
- *Using the debugger* on page 3-14
- *Displaying image information* on page 3-23.

## 3.1 The ADU desktop

The main features of the ADU desktop are:

- A menu bar, toolbar, mini toolbar and status bar. For details see *Menu bar, toolbar, mini toolbar and status bar* on page 3-3.
- A number of windows that display a variety of information as you work through the process of debugging your executable image. A window-specific menu is available for each window, as described in *Window-specific menus* on page 3-4.
- Three windows (Execution, Command, and Console) that are opened automatically when you start the debugger. The Execution window is always open. You can open other windows by selecting the appropriate item from the **View** menu. For details of all the available windows see *ADU desktop windows* on page 3-4.

Figure 3-2 shows ADU with the Execution, Console, Globals and Locals Windows, in the process of debugging the sample image `DHRY`.



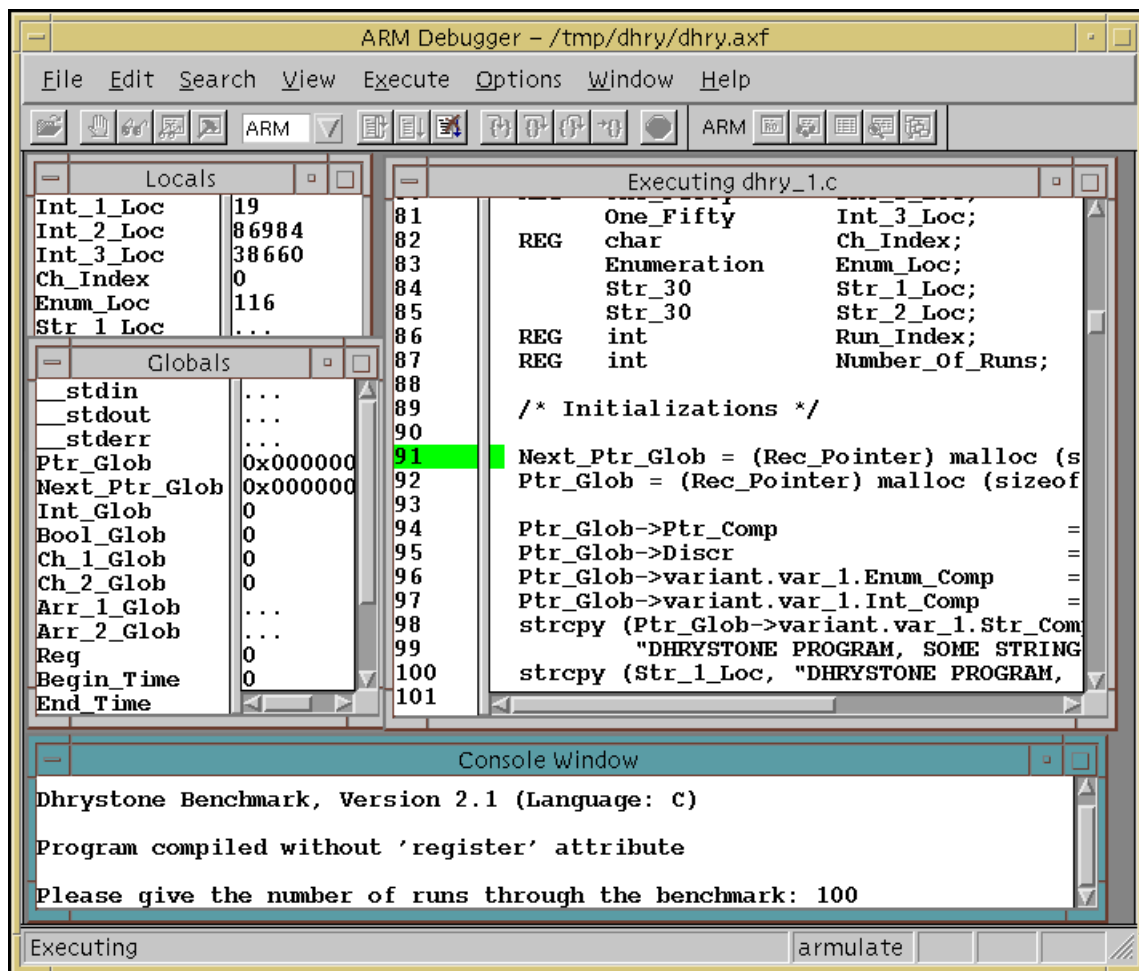


Figure 3-1: A typical ADU desktop display

### 3.1.1 Menu bar, toolbar, mini toolbar and status bar

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

Underneath the menu bar is the toolbar. Position the cursor over an icon and a brief description is displayed. There is also a processor-specific mini toolbar. The menu, 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.

### 3.1.2 Window-specific menus

Each of the 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.

Item-specific options require that you position the cursor over an item in the window before they are activated.

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

### 3.1.3 ADU desktop windows

The main ADU windows are always opened when you start ADU. They are the:

- Execution window
- Console window
- Command window.

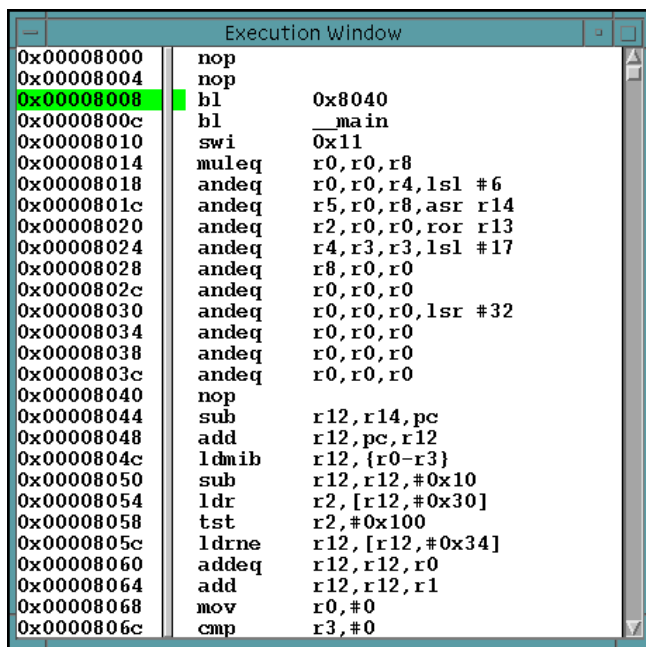
Use the **View** menu to display any of the following additional windows:

- Backtrace window
- Breakpoints window
- Debugger Internals window
- Disassembly window
- Expression window
- Function Names window
- Locals/Globals window
- Low Level Symbols window
- Memory window
- Registers window
- RDI Log window
- Search Paths window
- Source Files List window
- Source File window
- Watchpoints window.

The following sections describe the purpose of each window.

## Execution window

The Execution window (Figure 3-2) displays the source code of the program that is currently executing.



**Figure 3-2: Execution window**

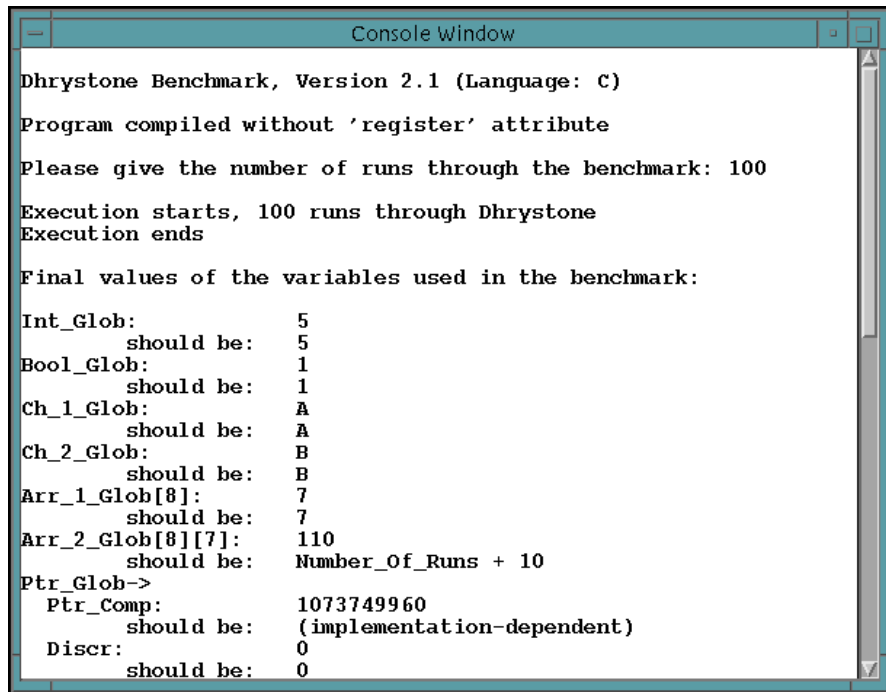
Use the Execution window to:

- 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
- set, edit, or remove breakpoints.

## Console window

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

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



```

Console Window

Dhrystone Benchmark, Version 2.1 (Language: C)

Program compiled without 'register' attribute

Please give the number of runs through the benchmark: 100

Execution starts, 100 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]: 110
    should be:    Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:        1073749960
    should be:     (implementation-dependent)
  Discr:           0
    should be:     0

```

**Figure 3-3: Console window**

Initially the Console window displays the start-up messages of your target processor, for example the ARMulator or ARM Development Board.

#### ———— Note ————

When input is required from the debugger keyboard by your executable image, most ADU functions are disabled until the required information has been entered.

## Command window

Use the Command window (Figure 3-4) to enter armsd instructions when you are debugging an image.

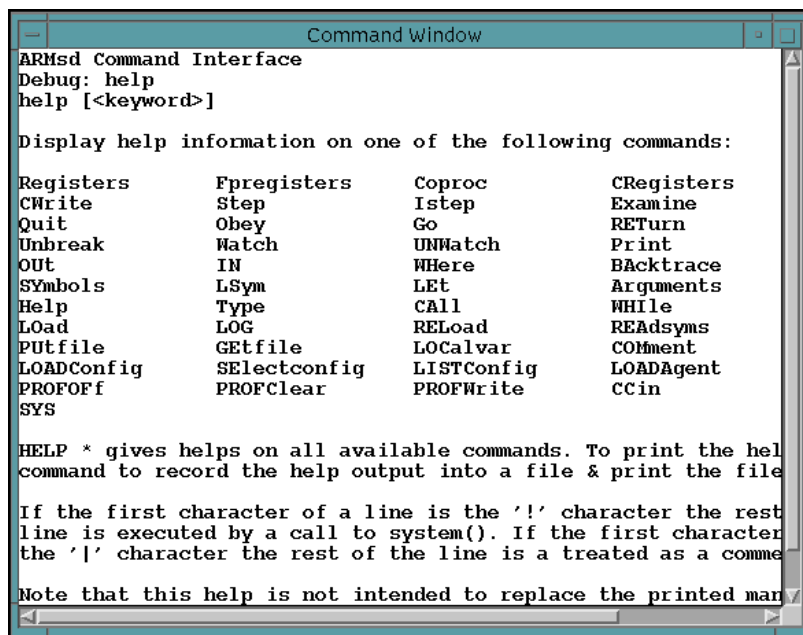


Figure 3-4: Command window

See *Using command-line debugger instructions* on page 4-9 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 the *ARM Software Development Toolkit Reference Guide*.

### Backtrace window

The Backtrace window displays current backtrace information for your program. Use the Backtrace window to:

- show disassembled code for the current procedure
- show a list of local variables for the current procedure
- set or remove breakpoints.

### 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/disassembly code
- edit or remove breakpoints.

## Debugger Internals window

This window displays some of the internal variables used by ADU. You can use this window to examine the values of the following variables, and to change the values of those not marked read-only:

<code>\$clock</code>	Number of microseconds elapsed since the application program began execution. This value is based on the ARMulator clock speed setting, and is unavailable if that speed is set to 0.00 (see also <i>ARMulator configuration</i> on page 5-9). This variable is read-only.
<code>\$cmdline</code>	Argument string for the image being debugged.
<code>\$echo</code>	Non zero if commands from obeyed files should be echoed (initially set to 0).
<code>\$examine_lines</code>	Default number of lines for the <code>examine</code> command (initially set to 8).
<code>\$format</code>	Default format for printing integer values (initially set to <code>%ld</code> ).
<code>\$fpresult</code>	Floating-point value returned by last called function (junk if none, or if a floating point value was not returned). This variable is read-only.
<code>\$inputbase</code>	Base for input of integer constants (initially set to 10).
<code>\$list_lines</code>	Default number of lines for the <code>list</code> command (initially set to 16).
<code>\$pr_linelength</code>	Default number of characters displayed on a single line (initially set to 72).

`$rdi_log` RDI logging (see Table 3-1):

Table 3-1: RDI logging

Bit 1	Bit 0	Meaning
0	0	Off
0	1	RDI on
1	0	Device Driver Logging on
1	1	RDI and Device Logging on

You can set these bits of the `$rdi_log` internal variable from the Debugger Internals window. For more information see *RDI Log window* on page 3-13 and *Remote debug information* on page 3-28.

`$result` Integer result returned by last called function (junk if none, or if an integer result was not returned). This variable is read-only.

`$sourcedir` Directory containing source code for the program being debugged.

`$statistics`  
This variable contains any statistics which the ARMulator has been keeping. You can examine the contents of this variable by clicking on `statistics` in the Debugger Internals window. This variable is read-only.

`$statistics_inc`  
Not applicable to ADU.

`$statistics_inc_w`  
This variable is similar to `$statistics`, but outputs the difference between the current statistics and the point at which you asked for the `$statistics_inc_w` window. To create a `$statistics_inc_w` window, select this item, right-click to display the menu, and select **Indirect through item**. This variable is read-only.

`$stop_of_memory`  
If you are using an EmbeddedICE interface, set this variable to the total amount of memory normally on the board. If you add more memory to the board, change this variable to reflect the new amount of memory.

`$type_lines`  
Default number of lines for the `type` command (initially set to 10).

\$vector\_catch

Applying to ARMulator and EmbeddedICE only, this determines which conditions are detected, with control passing back to the debugger. The default value is %RUsPDAifE. A capital letter indicates a condition that is intercepted:

R	reset
U	undefined instruction
S	SWI
P	prefetch abort
D	data abort
A	address
I	normal interrupt request (IRQ)
F	fast interrupt request (FIQ)
E	unused

## 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 more information on displaying disassembled code, see *Disassembled code* on page 3-27.

## 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 *Variables* on page 3-24.

## Function Names window

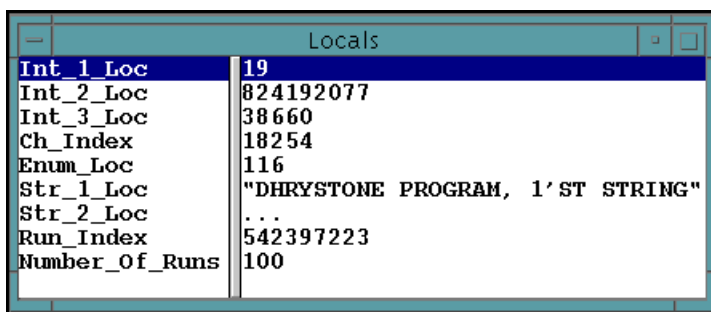
The Function Names window lists the functions that are part of your program.

Use the Function Names window to:

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

## Locals/Globals windows

The Locals window (Figure 3-5) displays a list of local 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.



**Figure 3-5: Locals window**

Use the Locals/Globals window to:

- change the content of a variable (double-click on it)
- display the section of memory pointed to by a variable
- change the format of the values displayed by line or for the entire window (if the format of a line is changed, it is no longer affected by changing the format of 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 *Variables* on page 3-24.

## 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.

## Memory window

The Memory window displays the contents of memory at 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).

You can have multiple memory windows open at any time.

## 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.

### ———— Note —————

Multiple register mode windows can be open at any one time, but no more than one window per mode. For example, you may open no more than one FIQ register window at a time.

## RDI Log window

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

---

### Note

---

This facility is not normally enabled. It must be specifically enabled when the RDI is compiled. In addition, the debugger internal variable `$rdi_log` must be non-zero.

---

For more information on RDI, see *Remote debug information* on page 3-28.

## 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.

## Source Files List window

The Source Files List window displays a list of all source files that are part of the loaded image.

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

## 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 *Source files* on page 3-23.

## Watchpoints window

The Watchpoints window displays a list of all watchpoints.

Use the Watchpoints window to:

- delete a watchpoint
- edit a watchpoint.

## 3.2 Using the debugger

This section explains how to:

- start up and close down the debugger
- load, reload, and execute an image
- step through an image
- set and remove breakpoints and watchpoints
- examine and set variables, registers, and memory.

For information on more advanced features, see the following:

- *Debugger configuration* on page 5-2
- *Displaying image information* on page 3-23
- *Further Debugging Features* on page 4-1
- ADU online help.

### 3.2.1 Starting up and closing down the debugger

Start up and close down ADU as follows:

#### Starting up ADU



Start up ADU in any 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
- launch ADU from APM or the DOS command-line, optionally with arguments.

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

- debug *ImageName*  
Load *ImageName* for debugging.
- exec *ImageName*  
Load and run *ImageName*.
- reset Reset the ADU 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`

Obeys the *ScriptName* on startup, this is the equivalent to typing `obey ScriptName` as soon as the debugger starts up. This works only for the ARMulator, not when a remote target, such as `Remote_A`, is selected.

For example, to launch ADU from the command-line and load `sorts.axf` for debugging, but without setting a breakpoint on `main()`, use:

```
adu -debug sorts.axf -nomainbreak
```

When you start ADU, the Console, Command, and Execution Windows are displayed, and you can go on to load your executable image.

### Closing down ADU

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

## 3.2.2 Loading, reloading, and executing a program image

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

### Loading an image



When you load a program image, the program is displayed in the Execution Window as disassembled code. The Command and Console Windows are also displayed. 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.

#### Note

After executing your program you must reload it to execute again. Use the **Reload** button or select **Reload** from the **File** menu (see *Reloading an image* on page 3-16)

1. Choose **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 wish to debug.
3. Enter any command-line arguments expected by your image.
4. Click **OK**.

Alternatively, if you have recently loaded your image, your file appears as a recently used file on the **File** menu.

See also *Flash download* on page 4-11.

---

**Note**

---

If you load your image from the recently used file list, ADU automatically loads the image using the command-line arguments you specified in the previous run.

---

## Reloading an image

After you have executed an image you must reload it before you can execute it again.



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

## Executing an image

You can run your program in ADU in either of the following ways:

- You can execute the entire program, with ADU halting execution at any breakpoints or watchpoints encountered, in either of the following ways:
  - Select **Go** from the **Execute** menu
  - Click the **Go** button.
- You can step through the code a line at a time, stepping into or over procedures as needed. This is described in the next section.



While the program executes, the Console Window is activated and the program code is displayed in the Execution Window.

Execution continues until:

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



You can then continue execution from the point where the program stopped using **Go** or **Step**.

---

**Note**

---

If you wish to execute your program again, you must reload it first.

---

### 3.2.3 Stepping through an image

To follow more closely the execution of a program image, you can step through the code in the following ways.

#### Step to the next line of code

Step to the 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 into a function call

Step into 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 as 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

Proceed as follows:



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.

---

### 3.2.4 Setting and removing breakpoints and watchpoints

Breakpoints and watchpoints are used to stop program execution when a selected line of code is about to be executed or when a specified condition occurs. Breakpoints and watchpoints may be either simple or complex. This section discusses simple breakpoints and watchpoints. Complex breakpoints and watchpoints are discussed in *Setting and editing complex breakpoints and watchpoints* on page 4-2.

#### Setting a simple breakpoint

There are 2 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. Press **OK** 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 either 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. If the line in which the breakpoint is set contains several functions, the breakpoint is set on the function on which you clicked.

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, but execution halts before `world()` is executed.

If you want to see all the breakpoints set in your program, open the Breakpoints Window by selecting **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. This method also works for the Low Level Symbols window, but the breakpoint is set to the first machine instruction of the function, that is, at the beginning of its entry sequence.

## Removing a simple breakpoint

There are four methods you can use to remove a simple breakpoint:

### Method 1

1. Double-click on a line containing a breakpoint (highlighted in red) in the Execution Window.
2. Click the **Delete** button from 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 resulting pop up menu.

### Method 3

1. Single click on a line containing a breakpoint (highlighted in red) in the Execution Window.
2. Click the **Toggle breakpoint** button from 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 wish to remove.
3. Click the **Toggle breakpoint** button or press the **Delete** key.

## 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 the **Toggle Watchpoint** option from the window-specific menu
  - click the **Watchpoint** button.



To see all the watchpoints set in your executable image, open the Watchpoints Window by selecting **Watchpoints** from the **View** menu.

## Removing a simple 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 wish 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 containing a watchpoint and right click.
2. Select **Toggle Watchpoint** from the 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.

## 3.2.5 Examining and modifying variables, registers, and memory

You can use the Debugger to display and modify the contents of the variables and registers used by your executable image. You can also examine the contents of memory. This section briefly introduces these features. See *Variables* on page 3-24 for more information on variables and registers, and *Saving or changing an area of memory* on page 4-6 for more information on working with areas of memory.

## Variables

You may display and modify either local or global variables.

To display a list of local variables:

1. Select the **Variables** submenu from the **View** menu.
2. Select **Local** from the **Variables** submenu.

An alternative method of displaying a list of local variables is:



1. Click the **Locals** button on the toolbar.

To display a list of global variables:

1. Select the **Variables** submenu from the **View** menu.
2. Select **Global** from the **Variables** submenu.

A Locals or Globals Window is displayed listing the variables currently active.

To modify the value of a variable :

1. Display either the Locals or the Globals window, as described above.
2. Double-click on the value you wish to change in the right pane of the window. The Modify Item dialog is displayed.
3. Enter the new value for the variable.
4. Click **OK**.

## Registers

To display a list of registers for the current processor mode:



1. Click the **Current Registers** button on the toolbar.

To display a list of 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 modify the value of a register :

1. Display an appropriate list of registers as described above.
2. Double-click on the register you wish to modify. The Modify Item dialog appears.
3. Enter the new value for the register.
4. Click **OK**.

## 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.
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 4 KB area of memory by using the scrollbar
- more remote areas of memory by entering another address.

To enter another address:

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.
3. Click **OK**.

## 3.3 Displaying image information

Various debugger windows are described in *ADU desktop windows* on page 3-4. This section gives more details of some of those windows, and describes other information that is also available to you during a debugging session.

### 3.3.1 Source files

You may view the paths that lead to the source files for your program image, list the names of the source files, or examine the contents of specific source files.

#### 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 may need to edit the search paths for your image manually. This depends on the options you chose when you built it.

If the files have been moved since the image was built, the new search paths for these files must be set up in the ARM Debugger, using the Add Path dialog (see below).

To display source file search paths:

1. 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 wish 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

To examine the source files of the current program:

1. Display the list of source files by selecting **Source Files** from the **View** menu. The Source Files List Window is displayed.
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.

---

### 3.3.2 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.

To display the value of a single variable:

1. Select **Expression** from the **Variables** submenu from the **View** menu.
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 **Immediate Evaluation** from **Variables** submenu from the **View** menu 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.

---

### Display formats

If the currently active window is the Locals, Globals, Expressions, or Debugger Internals window, you can change the format of a variable (see Table 3-2). Use the same syntax as a `printf()` format string in C to modify the display format for values of variables. Format descriptors include:

**Table 3-2: 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 an integer:	
	%c	Character
char*	%s	Pointer to character. Only use this for expressions which yield a pointer to a zero terminated string.
void*	%p	Pointer (same as % . 8x), eg. 00018abc. 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

To change the format of a variable:

1. Right click on the variable and select the **Change line format** from the Locals or Globals Window menu. The Display Format dialog is displayed.
2. Enter the display format.
3. Click **OK**.

---

**Note**

---

If you change a single line, that line is not affected by global changes.

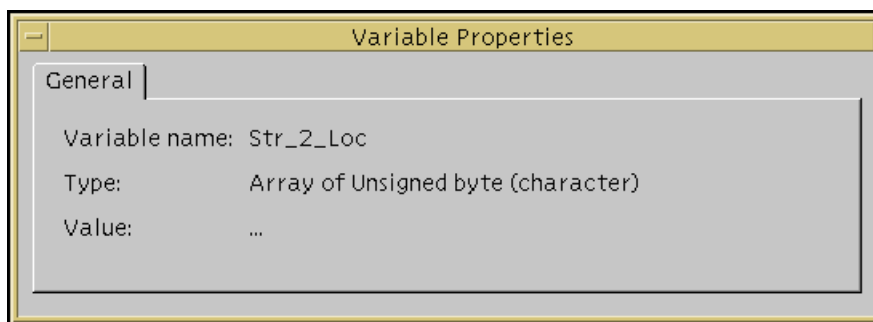
Leave the Display Format dialog empty and click **OK** to restore the default display format. Use this method to revert a line format change to the global format.

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

---

## 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 3-6). To display the window-specific menu, right click on an item. The information is displayed in a dialog.



**Figure 3-6: 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.



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

To display or hide disassembled code in the Execution window, select **Toggle Interleaving** from the **Options** menu.

Disassembled code is displayed in grey, the C or C++ code 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.
3. Click **OK**.

The Disassembly window displays the assembler instructions derived from the code held in the specified area of memory.

When you have opened the Disassembly window you can display the contents of another memory area as disassembled code by using the scroll bars to search for an address by value or:

1. Select **Goto** from the **Search** menu.
2. Enter an address.
3. Click **OK**.

#### Specifying a disassembly mode

The debugger tries to display disassembled code as ARM code or Thumb code as appropriate. Sometimes, however, the type of code required cannot be determined. This could happen, for example, if you have copied the contents of a disk file into memory.

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.

### 3.3.4 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/off

**Bit 1**            Device driver logging on/off

———— **Warning** ————

The RDI log level is used internally within ARM to assist with debugging, this level should be changed only if you have been requested to do so by ARM.

---

# Chapter 4

## Further Debugging Features

This chapter describes the features of ADU that are beyond the scope of the *Getting Started* chapter. It contains the following sections:

- *Setting and editing complex breakpoints and watchpoints* on page 4-2
- *Saving or changing an area of memory* on page 4-6
- *Specifying command-line arguments for your program* on page 4-8
- *Using command-line debugger instructions* on page 4-9
- *Profiling* on page 4-10
- *Flash download* on page 4-11.

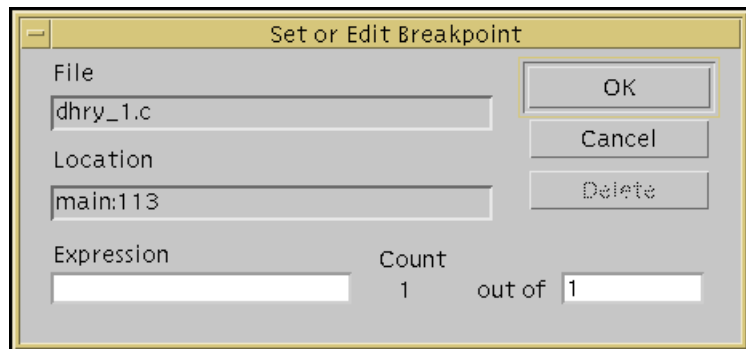
## 4.1 Setting and editing complex breakpoints and watchpoints

When you set a complex breakpoint or watchpoint, you specify additional conditions in the form of expressions entered in a dialog.

### 4.1.1 Breakpoints

A breakpoint is a point in the code where your program is halted by the ARM Debugger. When you set a breakpoint it appears as a red marker in the left-hand pane of a Source or Execution window.

When you set a complex breakpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Breakpoint dialog (Figure 4-1).



**Figure 4-1: 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 hex 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.

### Setting or editing a complex breakpoint

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 highlighted 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. Enter or modify 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. Enter or modify the details of the breakpoint.
4. Click **OK**.

#### 4.1.2 Watchpoints

A watchpoint halts a program when a specified register or variable is changed.

When you set a complex watchpoint, you specify additional conditions in the form of expressions entered in the Set or Edit Watchpoint dialog (Figure 4-2).

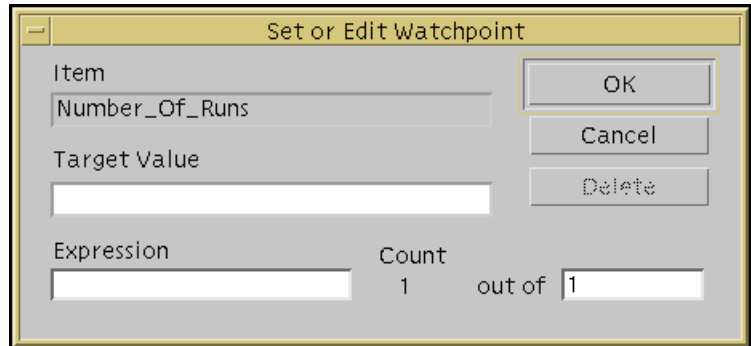


Figure 4-2: Set or Edit Watchpoint dialog

This dialog contains the following fields:

**Item** The variable or register to be watched.

**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** Any expression that must be true for the program to halt, in addition to any other watchpoint conditions. As with breakpoints, use C-like operators such as:

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

**Count** The program halts when all 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. The Set or Edit Watchpoint dialog is displayed.
4. Specify the details of the watchpoint.
5. 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.
3. Modify the details as required.
4. Click **OK**.

## 4.2 Saving or changing an area of memory

You can either copy an area of memory to a disk file or copy the contents of a disk file to an area of memory.

### 4.2.1 Saving an area of memory to a file on disk

To save an area of memory to a file on disk:

1. Select **Put File** from the **File** menu. The Put file dialog is displayed.

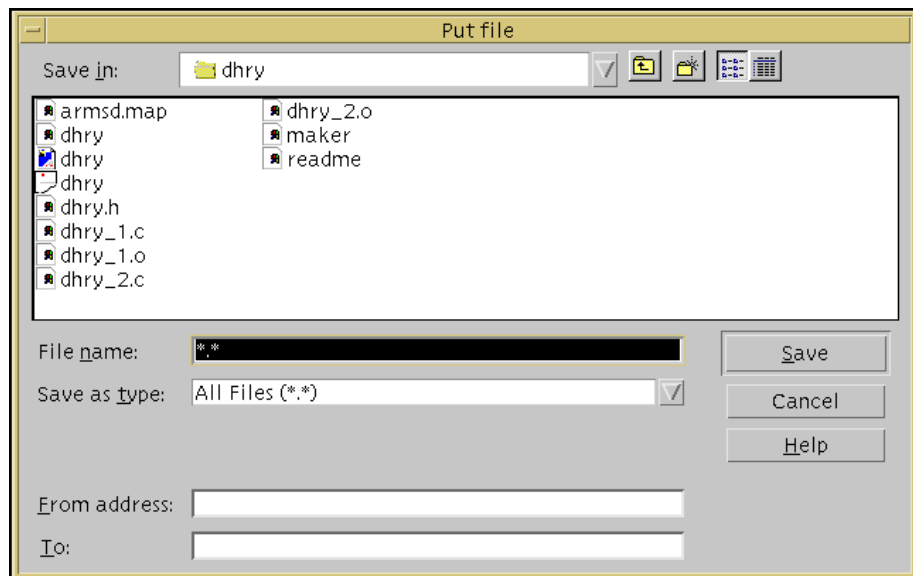


Figure 4-3: Put file dialog

2. Select the file to write to.
3. Enter a memory area in the **From address** and **To** fields.
4. Click **Save**.
5. Click **OK**.

———— **Note** ————

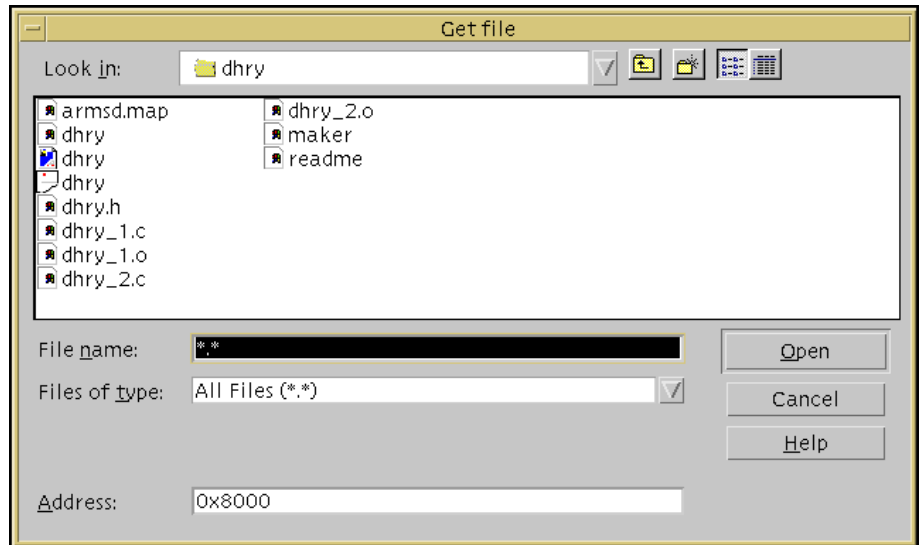
The output is saved as a binary data file.

### 4.2.2 Copying a file on disk to memory

To copy the contents of a disk file to memory:



1. Select **Get File** from the **File** menu. The Get file dialog is displayed.

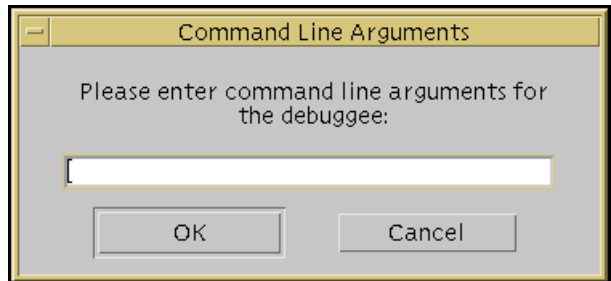


**Figure 4-4: Get file dialog**

2. Select the file you want to load into memory.
3. Enter a memory address where the file should be loaded.
4. Click **Open**.

### 4.3 Specifying command-line arguments for your program

1. Select **Set Command Line Args** from the **Options** menu. The Command Line Arguments dialog is displayed.



**Figure 4-5: 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 Load Image dialog or by changing the Debugger internal variable, `$cmdline`.

## 4.4 Using command-line debugger instructions

If you are familiar with the ARM symbolic debugger (armsd) you may prefer to use the same set of commands from the Command Window.

Select **Command** from the **View** menu to open this window.

The Command window displays a Debug: command line. You can enter ARM command line debugger commands at this prompt. The syntax used is the same as that used for armsd. Type **help** for information on the available commands.

Refer to Chapter 4 of the *ARM Software Development Toolkit User Guide* and the *ARM Software Development Toolkit Reference Guide* for more information on armsd.

### 4.4.1 Bytes, shorts and words

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 length of data read or written, and how to change it if necessary. By default, a read from or write to memory in armsd or ADU transfers a word value of 4 bytes. 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
```

and 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 half-word ‘shorts’ in a similar way, for example:

```
let *(short *) 0xaddress = value
print /%x *(short *) 0xaddress
```

You can also select **View->Variables->Expression** to obtain the View Expression window, and use that to specify bytes or shorts for displaying memory.

For example, for bytes, enter `*(char *) 0xaddress` in the **View Expression** box., and for shorts, enter `*(short *) 0xaddress` in the **View Expression** box.

To display in hex, right-mouse click on the Expression window, select **Change Window Format** and enter %x.

## 4.5 Profiling

Profiling is available if you are using ARMulator or Angel. It allows you to see where most of the processor time is spent within a program, by sampling the pc at specified time intervals. This information is used to build up a picture of the percentage of time spent in each procedure. Using the command-line program armprof on the data generated by either armsd or ADU, you see where effort can be most effectively spent to make the program more efficient.

To collect profiling information:

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

---

### Note

---

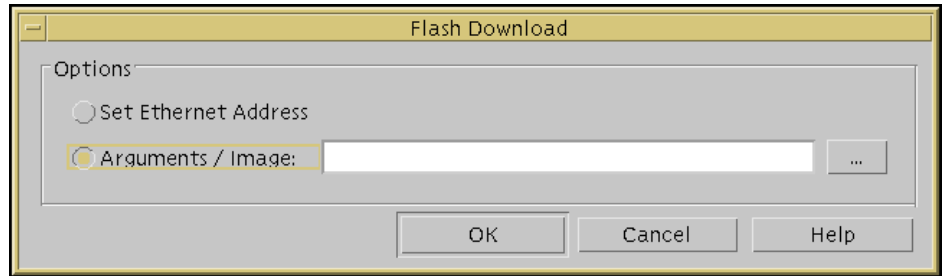
After you have started program execution you cannot turn profile collection on. However, if you want to collect information on only a certain part of the execution, you can initiate collection before executing the program, clear the information collected up to a certain point, such as a breakpoint, by selecting **Clear Collected** from the **Profiling** submenu off the **Options** menu, then execute the remainder of your program.

---

See Chapter 8 of the *ARM Software Development Toolkit User Guide* for more information on profiling.

## 4.6 Flash download

Use the Flash Download dialog to write an image to the flash memory on an ARM Development Board or any suitably equipped hardware.



**Figure 4-6: 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 *Target Development System User Guide*.



# Chapter 5

## Configurations

This chapter explains how you can change various configuration settings. It contains the following sections:

- *Debugger configuration* on page 5-2
- *ARMulator configuration* on page 5-9
- *Remote target configuration* on page 5-12
- *EmbeddedICE configuration* on page 5-14.

## 5.1 Debugger configuration

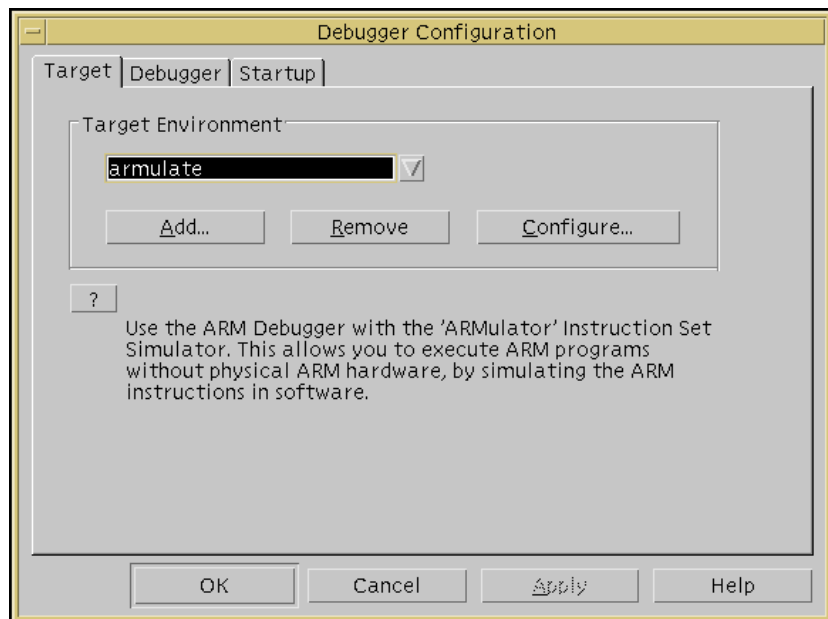
The Debugger Configuration dialog consists of three tabbed screens:

- Target
- Debugger
- Startup.

These are described below. Access the Debugger Configuration dialog by selecting **Configure Debugger** from the **Options** menu.

### 5.1.1 Target environment

Use the **Target** tab of the Debugger Configuration dialog (see Figure 5-1) to change the configuration used by the target environments to be used during debugging.



**Figure 5-1: Configuration of target environment**

#### Target Environment

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.

When you have changed any configuration settings for the target environment, you can either make the new settings effective or ignore them, as follows:

- click **OK** to save any changes and exit
- click **Apply** to save any changes
- click **Cancel** to ignore all changes not applied and exit.

---

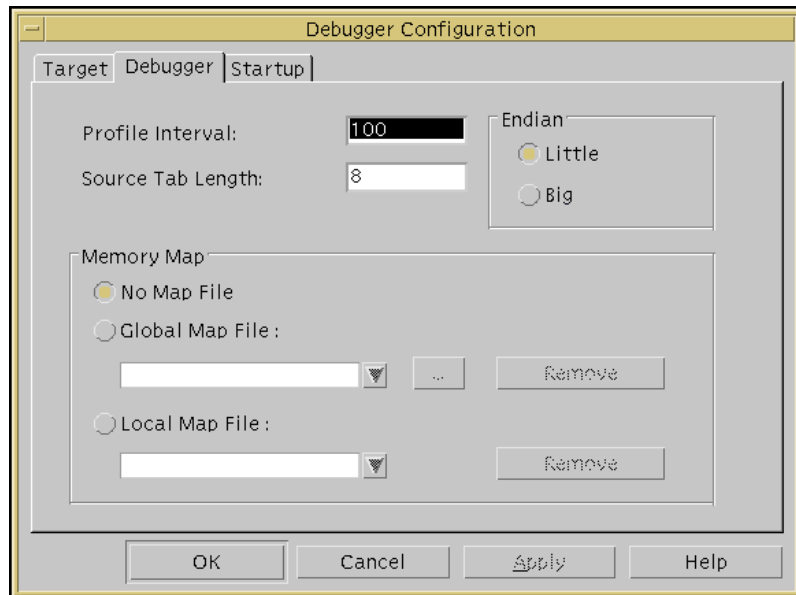
**Note**

**Apply** is disabled for the Target page because a successful RDI connection has to 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.

---

### 5.1.2 Debugger

Use the **Debugger** tab of the Debugger Configuration dialog (see Figure 5-2 on page 5-4) to change the configuration used by the Debugger.



**Figure 5-2: Configuration of debugger**

With the **Debugger** tab of the Debugger Configuration dialog active, you can examine and change the following configuration settings:

#### **Profile Interval**

This is the time between pc-sampling in microseconds. Applicable to ARMulator and Angel only. Lower values have a higher performance overhead, and slow down execution. Higher values are not as accurate as lower values.

#### **Source Tab Length**

This specifies the number of space characters used for tabs when displaying source files.

#### **Endian**

Determines byte sex of the target.

**Little** low addresses have the least significant bytes.

**Big** high addresses have the least significant bytes.

## Memory Map

This allows you to specify a memory map file, containing information about a simulated memory map that the ARMulator uses. Applicable to ARMulator only. The file includes details of the databus widths and access times for each memory region in the simulated system. For further information, see Chapter 8 of the *ARM Software Development Toolkit User Guide*.

You can select one of three Memory Map options:

- do not use a memory map
- use a global memory map, which means using the specified memory map for every image that is loaded during the current debug session
- use a local memory map, which means using a memory map that is local to a project.

These Memory Map options are described in detail below.

When you have finished making changes to the configuration settings to be used by the debugger, you can either make the changes effective or ignore them, as follows:

- 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.

---

The three Memory Map options are explained in greater detail as follows:

### No Map File

Select this Memory Map option to use the ARMulator default memory map. This is a flat 4GB bank of 'ideal' 32-bit memory, having no wait states.

### Global Map File

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

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 an image loaded, the directory containing the image is searched.

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. Do not specify an *absolute* path name. You must specify a memory map file *relative* to the current image path.

This option has no browse button because the local memory map file is path independent. The debugger searches for the local memory map file in directories as described above.

When you have selected a filename, or typed in 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.

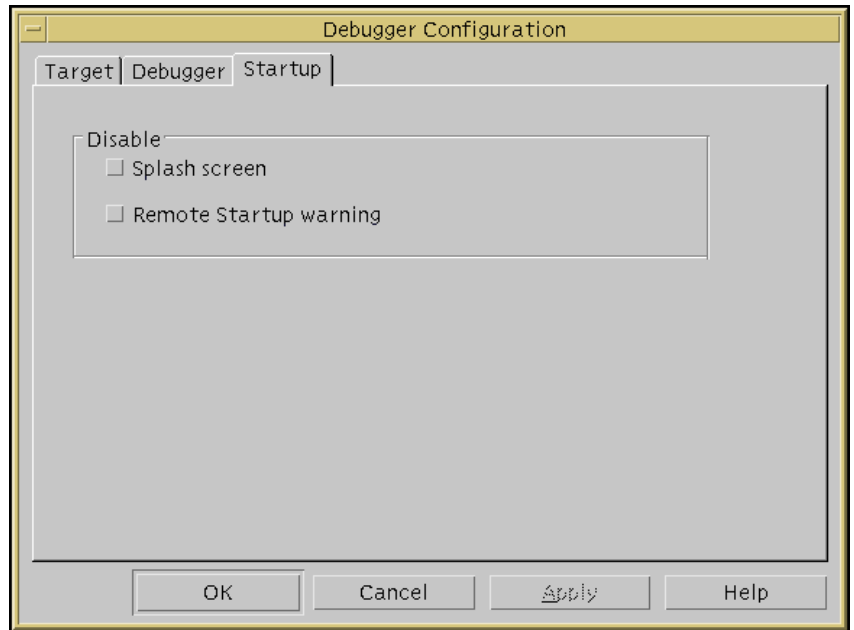
Use the **Remove** button to remove the currently selected file from the list.

### ———— 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 the **Configure Debugger** option from the **Options** menu and clicking **OK**.

### 5.1.3 Startup

Use the **Startup** tab of the Debugger Configuration dialog (see Figure 5-3) to configure the ADU startup process.



**Figure 5-3: Configuration of the ADU startup process**

With the Startup tab of the Debugger Configuration dialog active, you can examine and change the following configuration settings:

#### **Disable Splash screen**

When selected, stops display of the splash screen (the ADU startup box) when ADU is first loaded.

#### **Disable 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 ADU may hang. If the warning is enabled, you have the opportunity to start in ARMulate.

When you have finished making changes to the configuration settings used when ADU starts up, you can either make the changes effective or ignore them, as follows:

- click **OK** to save any changes and exit

- click **Apply** to save any changes
- click **Cancel** to ignore all changes not applied and exit.

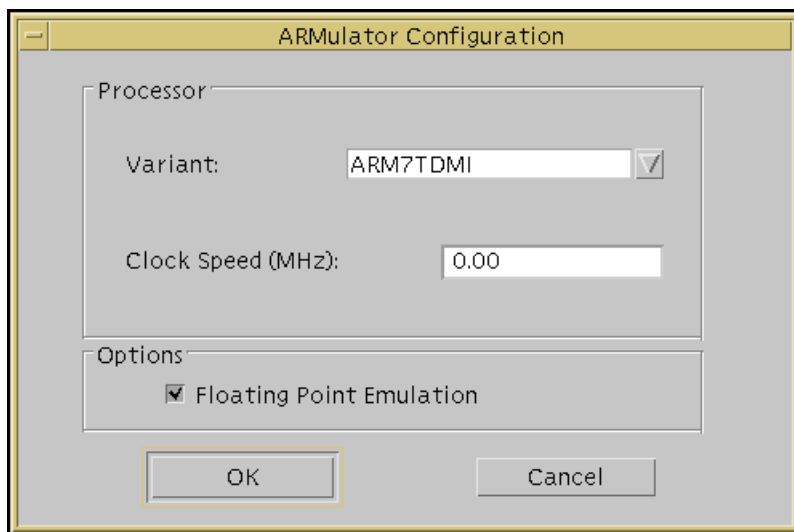
## 5.2 ARMulator configuration

Use the **ARMulator Configuration** dialog to change configuration settings for the ARMulator.

To access this dialog:

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.

The ARMulator Configuration dialog is displayed (see Figure 5-4).



**Figure 5-4: Configuration of ARMulator**

With the ARMulator Configuration dialog active, you can examine and change the following configuration settings:

**Variant** Processor type required for emulation.

**Clock speed** Clock speed to be used for emulation. Values stored in debugger internal variable `$clock` depend on this setting, and are unavailable if you specify a clock speed of 0.00 (see *Debugger Internals* window on page 3-8).

The ADU clock speed initially defaults to 0.00 for compatibility with the defaults of `armsd`. A clock speed of zero in ADU is equivalent to omitting the `-clock` `armsd` option on the command-line. In other words, a ‘zero’ clock frequency really means ‘unspecified’.

For the ARMulator, an unspecified clock frequency is of no consequence because ARMulator does not need a clock frequency to be able to ‘execute’ instructions and count cycles (for `$statistics`).

However, your application program may sometimes need to access a clock, for example, if it contains calls to the standard C function `clock()` or the Angel `SWI_clock`, so ARMulator must always be able to give clock information. It does so in the following way:

- if a clock speed has been specified to ADU or `armsd`, then ARMulator uses that frequency value for its timing
- if the clock speed is zero (for ADU) or unspecified (for `armsd`), the real-time clock of the host computer is used by ARMulator instead of an emulated clock.

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 Angel `SWI_clock`, and is also visible to the user from the debugger as `$clock`. It is also used internally by ADW and `armsd` in the calculation of `$memstats`.

Note that 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.

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

However, if no clock speed is specified (i.e. 0.00MHz) then, for the purposes of calculating the wait-states, an arbitrary default of 1MHz is used to calculate a core:memory clock ratio, so that `$memstats` can still give useful memory timings.

### **Floating point emulation**

Toggles floating point emulation on and off.



If you are using the software floating-point C libraries, ensure that this option is off (blank). The option should be on (checked) only if you are using the hardware floating-point emulator (FPE).

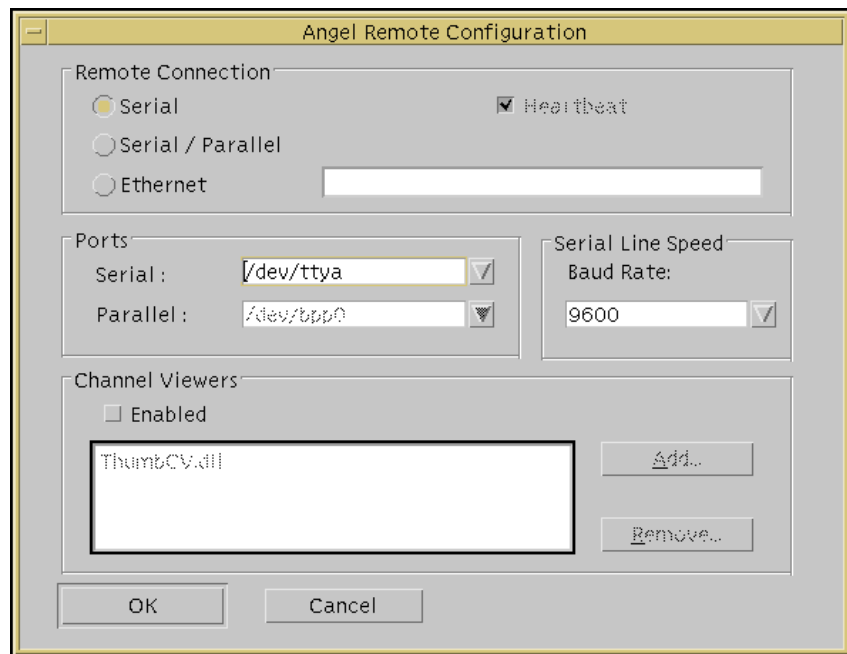
## 5.3 Remote target configuration

If you are using Angel or the EmbeddedICE interface, use the Angel Remote Configuration dialog to configure the settings for the Remote\_A connection you are using to debug your application.

To access this dialog:

1. Select **Configure Debugger** from the **Options** menu.
2. Click on the **Target** tab.
3. Select **Remote\_A** in the Target Environment field to select ADP (Angel Debug Protocol).
4. Click on the **Configure** button.

The Angel Remote Configuration dialog is displayed (see Figure 5-5).



**Figure 5-5: Configuration of Remote\_A connection**

With the Angel Remote Configuration dialog active, you can examine and change the following configuration settings:

**Remote Connection**

Chooses either Serial or Serial/Parallel depending on the connections used by your development board. For Ethernet, enter either an IP address or the hostname of the target board.

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

**Ports** Allows the correct serial and parallel devices to be chosen for the debug connection.

**Serial Line Speed**

Selects the Baud rate used to transmit data along the serial line.

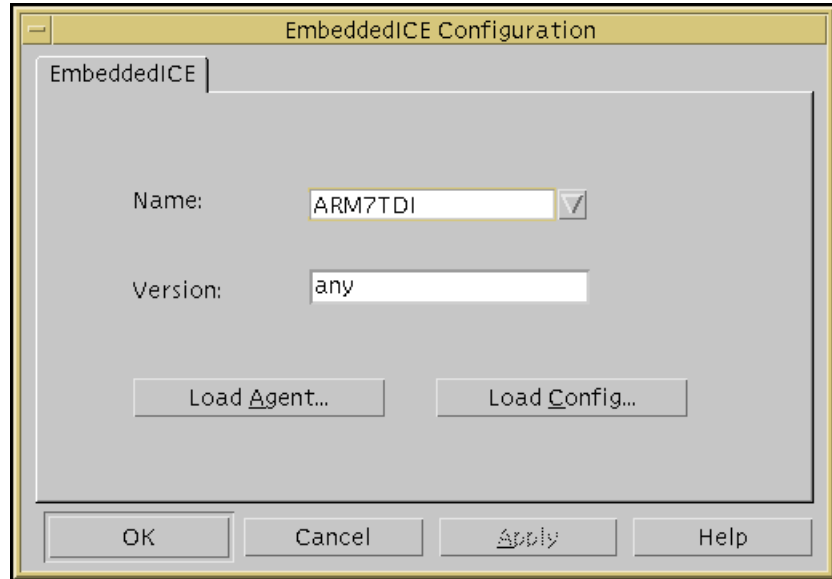
**Channel Viewers**

Channel viewers are not supported by ARM Debugger for UNIX.

## 5.4 EmbeddedICE configuration

Use the EmbeddedICE Configuration dialog to select the settings for EmbeddedICE. This option applies only if you are using EmbeddedICE.

Select **Configure EmbeddedICE** from the **Options** menu to access this dialog (see Figure 5-6).



**Figure 5-6: Configuration of EmbeddedICE target**

With the EmbeddedICE Configuration dialog active, you can examine and change the following configuration settings:

- |                |  |
|----------------|--|
| <b>Name</b>    | Name given to the EmbeddedICE configuration. Valid options are:<br><b>ARM7DI</b> for use with an ARM7 core with debug extensions and EmbeddedICE macrocell (includes ARM7DMI)<br><b>ARM7TDI</b> for use with an ARM7 core with Thumb and debug extensions and EmbeddedICE macrocell (includes ARM7TDMI). |
| <b>Version</b> | Version given to the EmbeddedICE configuration. Specify the specific 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 major updates to the ROM.
- Load Config** Specify an EmbeddedICE configuration file to be loaded. Click OK to run. Use this for minor updates.



# Appendix A

## FlexLM License Manager

This appendix describes the use made by ARM Limited of FlexLM license management software. It contains the following sections:

- *About license management* on page A-2
- *Obtaining your license file* on page A-4
- *What to do with your license file* on page A-5
- *Starting the server software* on page A-6
- *Running your licensed software* on page A-7
- *Customizing your license file* on page A-9
- *Finding a license* on page A-11
- *Using FlexLM with more than one product* on page A-12
- *FlexLM license management utilities* on page A-14
- *Frequently asked questions about licensing* on page A-18

## A.1 About license management

FlexLM is a software licensing package that controls the usage of licensed software applications. Licensing is controlled by means of a license file that describes the software you may use and how many copies of it you may run concurrently.

You must obtain a valid license file from ARM Limited before you can run licensed ARM software. *Obtaining your license file* on page A-4 describes how to apply for your license file.

You must specify one or more computers to act as a license server, on which license management software runs. Any computer running FlexLM licensed software must either be a license server or have access to a license server.

ARM Debugger for UNIX is one example of software that requires a license server before you can run it.

The license server can be any one of:

- your local machine
- a remote machine
- several remote machines.

If you choose to use more than one, you must use three license server machines. These communicate with one another, and co-ordinate the licensing. The advantage of this is that if one of the license server machines fails to operate correctly the other two will continue to allow licensed software to be used. This arrangement is known as a *3-server redundant set*.

Remote license servers do not need to be running on the same hardware platform as the software they are controlling.

### A.1.1 Installing FlexLM software

License management software for various platforms is supplied on the ADU CD-ROM. The following list shows the platforms supported, and the subdirectory containing the appropriate software for each:

**Solaris 2.5** flexlm/solaris

**SunOS 4.1.x** flexlm/sunos

**HP-UX 9.x** flexlm/hpux

Each directory contains the software in TAR file format, in a file called `flexlm.tar`.

Before applying for a license file you must install the FlexLM license management software, as follows:



1. Copy the TAR file from the appropriate directory onto each license server machine.
2. On each license server machine, unTAR the file using the command:  

```
tar xvf flexlm.tar
```
3. When you have unTARed the software you need to run the `makelinks.sh` script. Change into the directory containing the unTARed software and type:  

```
./makelinks.sh
```
4. This creates numerous hard links, one of which is `lmhostid`.

You need `lmhostid` when you complete your license request form.

## A.2 Obtaining your license file

The installation directory contains a file called `license_request_form.txt`

To obtain your license file:

1. Open this text file with the editor of your choice.
2. Complete the form, following any instructions that are in the file. You must decide whether your license server is to be your local machine, a remote machine, or three machines:
  - To use your local machine as the license server, fill in the license request form with the hostname and hostID of your machine.
  - To use a remote machine as the license server, fill in the license request form with the hostname and hostID of the remote machine. Sometimes an organization will designate one machine as the machine to run all license servers, so find out if this is what happens in your company.
  - To specify three separate machines as license servers, fill in the hostname and hostIDs of all three machines on the license request form.
3. Return the form to ARM Limited, as follows:
  - if you have email available, paste the completed form into your email composition tool and send it using the email address contained within the template
  - if you do not have email available, print out the completed form and send a facsimile using the Fax number contained within the template.
4. A license file will be returned to you shortly.

## A.3 What to do with your license file

Make a copy of the license file on each of your license servers, as follows:

1. If you receive the license file by email you can either copy the license file section out of the message, or save the entire message to disk. The license server ignores all lines except those start with SERVER, VENDOR, or FEATURE.
2. If you received the license file by fax you will need to create a text file and key in the information, using the editor of your choice. When data entry is complete, you can use the `lmchecksum` utility to check that you typed everything in correctly. Instructions for using `lmchecksum` are given under *FlexLM license management utilities* on page A-14.
3. You may save the license file in any directory on each license server. It should, however, be on a locally mounted file system.
4. You usually need to edit the VENDOR line of the license file on each license server. The default license file sent you contains:

```
VENDOR armlmd /opt/arm/flexlm/solaris
```

Change the text `/opt/arm/flexlm/solaris` so that it specifies the directory that holds your license server software. Specifically, the directory that holds file `armlmd`.

5. Remember to do this on each license server.

Full instructions for editing the license file can be found under *Customizing your license file* on page A-9.

## A.4 Starting the server software

To start the license server software on each machine, go to the directory containing the license server software and type:

```
nohup lmgrd -c license_file_name -l logfile_name &
```

where:

*license\_file\_name*

specifies the fully qualified pathname of the license file

*logfile\_name*

specifies the fully qualified pathname to a log file.

When you have started the license server, you can type:

```
cat logfile_name
```

to see the output from the license server.

## A.5 Running your licensed software

Before you run your licensed software for the first time, you must set the environment variable `ARMLMD_LICENSE_FILE` to an appropriate value.

### A.5.1 Setting the environment variable `ARMLMD_LICENSE_FILE`

The required value depends on your circumstances, as follows:

#### **You have only one license server (option 1)**

Assuming your license server is called `enterprise`, go to the machine where the ARM Debugger is installed and type:

```
setenv ARMLMD_LICENSE_FILE @enterprise
```

#### **You have only one license server (option 2)**

Assuming your license file is called `arm-debugger.lic` and is in the directory `/home/licenses`, type:

```
setenv ARMLMD_LICENSE_FILE /home/licenses
```

#### **You have only one license server and specified a TCP port (option 1)**

Assuming your license server is called `enterprise` and you have specified TCP port 7117 in your license file, go to the machine where your licensed software is installed and type:

```
setenv ARMLMD_LICENSE_FILE 7117@enterprise
```

#### **You have only one license server and specified a TCP port (option 2)**

Assuming your license file is called `mylicense.txt` and is in the directory `/local/home/license`, type:

```
setenv ARMLMD_LICENSE_FILE /local/home/license/mylicense.txt
```

#### **You have 3 license servers**

Assuming your license file is called `license.lic` and is stored in the local directory `/opt/arm/licenses`, type either one of the following two commands:

```
setenv ARMLMD_LICENSE_FILE /opt/arm/licenses
```

```
setenv ARMLMD_LICENSE_FILE /opt/arm/licenses/license.lic
```

## **A.5.2 Running your application**

When you have set the environment variable `ARMLMD_LICENSE_FILE` to a suitable value, as described above, you can run your licensed software.

## A.6 Customizing your license file

Your license file contains information similar to that shown in one of the following examples:

### Example 1-1: Typical 1-server license file

---

```
SERVER jupiter 80826d02
VENDOR armlmd /opt/arm/flexlm/solaris
FEATURE adu armlmd 1.000 01-jan-1999 4 5B7E20C1A2338616F456 ck=42
```

---

### Example 1-2: Typical 3-server license file

---

```
SERVER jupiter 80826d02 7117
SERVER saturn 80af8111 7117
SERVER uranus 81873622 7117
VENDOR armlmd /opt/arm/flexlm/solaris
FEATURE adu armlmd 1.000 01-jan-1999 4 5B7E20C1A2338616F456 ck=42
```

---

Although you must not change FEATURE lines, you may need to change the SERVER and VENDOR lines in your license file.

### A.6.1 Server and Vendor lines

You may need to change SERVER and VENDOR lines for the following reasons:

#### Hostname on Server line

On occasion you may need to change the hostname of a license server. In such a case you must change the hostname in all copies of the license file that refer to that server.

If you supplied three hostnames on the license request form then there are three server lines in the license file.

#### TCP port on Server line

It is possible to specify on a SERVER line the TCP port that the license manager uses to communicate with the licensed software. If not specified the license manager will use the next available port in the range 27000–27009. When connecting to a server, an application tries all the ports in the range 27000–27009.

A port number must be specified on each SERVER line if a 3-server license is in use.

### Daemon path on Vendor line

On a VENDOR line you may need to change the second parameter, the pathname of the vendor daemon executable. This pathname must point to the directory containing file `armlmd`.

If the license server was running on a SunOS machine then the Vendor line could be similar to:

```
VENDOR armlmd /opt/arm/flexlm/sunos
```

## A.6.2 Feature lines

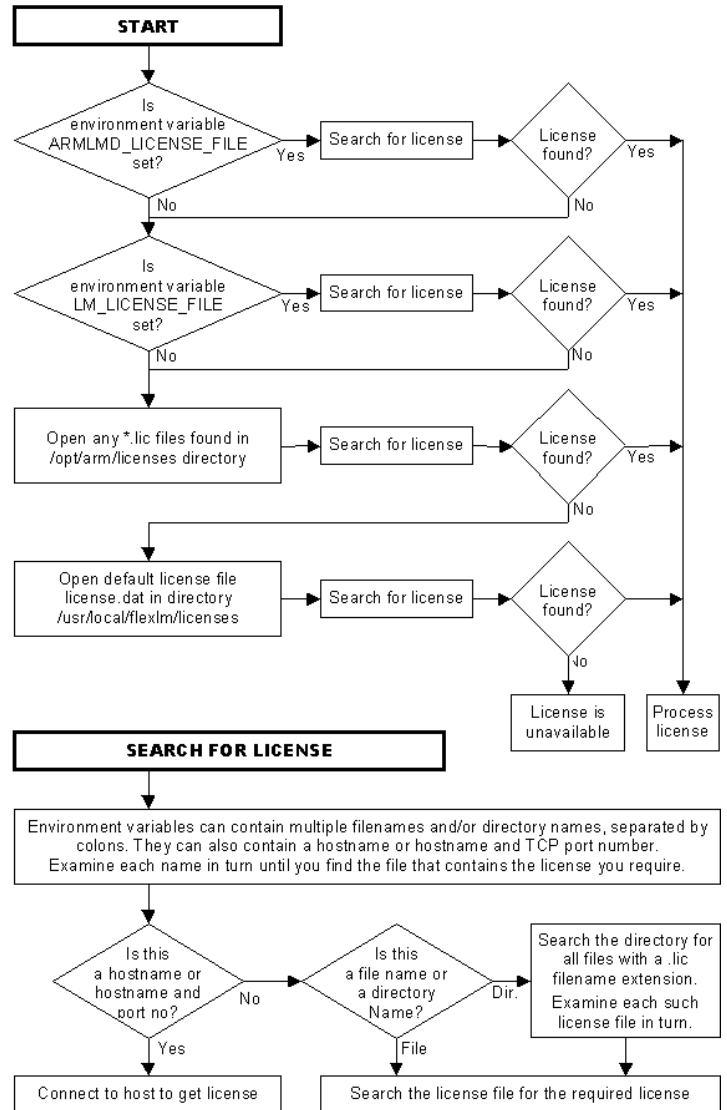
FEATURE lines describe the licenses that are available, and *must not* be altered. If they are altered the license is invalidated, and the feature no longer operates.

Each FEATURE line specifies the feature name, the vendor daemon name, the feature version, the expiration date of the license (a year of 0 means the license never expires), the number of concurrent licenses available, and the license key.



## A.7 Finding a license

The following diagram shows the rules followed by licensed software when it searches for a license authorizing it to run:



**Figure A-1: Finding a license**

## A.8 Using FlexLM with more than one product

FlexLM is a widely used product for license management, so it is possible that you have more than one product using FlexLM.

The latest version of the FlexLM software will always work with vendor daemons built using previous versions. Consequently you must always use the latest version of `lmgrd` and the FlexLM utilities.

---

### **Note**

---

The FlexLM software currently shipped by ARM is FlexLM version 6.0.

---

If you have multiple products using FlexLM you may encounter two situations:

- all the products use the same license server
- all the products use different license servers.

### A.8.1 All products use the same server

If the license files for every product contain exactly the same Server lines, ignoring different TCP port numbers, then there are two possible solutions:

1. Start a separate `lmgrd` daemon for each license file. There are no real disadvantages with this approach, as the separate daemons consume very little system resources or CPU time.
2. Combine the license files together. Take the `SERVER` line from one of the license files then add all of the other lines, that is the `DAEMON/VENDOR` and `FEATURE` lines, to create a new license file.

You will need to store the new combined license file in:

```
/usr/local/flexlm/licenses/license.lic
```

or give its location via the `LM_LICENSE_FILE` environment variable.

3. Start `lmgrd` using the new license file. Remember that you must use the latest version of `lmgrd` that is used by any of the products. You can use the command `lmgrd -v` or `lmver lmgrd` to find out the version of each `lmgrd`.

If the version of `lmgrd` is earlier than any of the vendor daemons, you see error reports such as:

```
Vendor daemon cannot talk to lmgrd (invalid data returned  
from license server)
```

Leave a symbolic link to the new license file in all the locations which held the original license files.

### A.8.2 All products use different license servers

If all the products use different hosts to run the license managers, then you must keep separate license files for each product.

Set the `LM_LICENSE_FILE` environment variable to point to the locations of all the license files, for example:

```
setenv LM_LICENSE_FILE license_file1:license_file2:  
...:license_filen
```

---

#### Note

FlexLM version 6.0 allows each software vendor to have an individual environment variable for finding the license file for their products. The environment variable name is `xxx_LICENSE_FILE` where `xxx` is the name of the vendor license daemon. In the case of software from ARM Limited the vendor daemon is called `ARMLMD`, therefore the environment variable for ARM software is `ARMLMD_LICENSE_FILE`. FlexLM version 6.0 vendor daemons always look for the vendor specific environment variable, ahead of the `LM_LICENSE_FILE` environment variable.

---

## A.9 FlexLM license management utilities

The **flexlm** directory on your product CD-ROM contains subdirectories holding the license manager utilities and the ARM vendor daemon (**armlmd**) for various platforms.

*Installing FlexLM software* on page A-2 describes how to install the software on your (one or three) license server machines.

The installation process creates a series of hard links to make your usage of the license management tools easier. Specifically it allows you to execute the utilities by using their short names, for example you can type **lmver** instead of **lmutil lmver**.

All the license tools are actually contained within the single executable **lmutil**, the behavior of which is determined by the value of its **argv[0]**.

### A.9.1 License administration tools

The **lmdown**, **lmremove**, and **lmreread** commands are privileged. If you started **lmgrd** with the **-p 2** switch then you must be a license administrator to run any of these three utilities.

A license administrator is a member of the UNIX **lmadmin** group or, if that group does not exist, a member of group 0.

In addition, **lmgrd -x** can disable **lmdown** and/or **lmremove**.

All utilities take the following arguments:

- v**                    print version and exit.
- c license\_file**                    operate on a specific license file.

#### **lmchecksum**

```
lmchecksum [-k] [-c license_file_name]
```

The **lmchecksum** utility performs a checksum of a license file. Use it to check for data entry errors in your license file. **lmcksum** prints a line-by-line checksum for the file as well as an overall file checksum. If the license file contains **cksum=nn** attributes, the bad lines are indicated automatically.

This utility is particularly useful if you received your license by Fax and typed the file, because of the possibility of data entry errors.

Use the **-k** switch to force the checksum to be case-sensitive.

By default `lmchecksum` checks the contents of `license.dat` in the current directory. Use the `-c` switch to check a different file.

## lmdiag

```
lmdiag [-c license_file_list] [-n] [feature]
```

This utility allows you to check for problems, when you cannot check out a license.

`-c license_file_list`

Path to file(s) to check. If more than one file, use a colon separator.

`-n` Run in non-interactive mode.

`feature` Diagnose this feature only. If you do not specify a feature, all lines of the license file are checked.

The `lmdiag` program first tries to check the feature. If this fails, the reason for failure is printed.

If the check failed because `lmdiag` could not connect to the license server then you can run extended connection diagnostics. These diagnostics try to check the validity of the port number in the license file. `lmdiag` displays the port numbers of all ports that are listening, and indicates which ones are `lmgrd` processes. If `lmdiag` finds the `armlmd` daemon for the for feature being tested, it displays the correct port number to use in the license file.

## lmdown

```
lmdown [-c license_file_list] [-vendor name] [-q]
```

The program allows you to shut down gracefully all license daemons on all nodes (both `lmgrd` and all vendor daemons).

`-c license_file_list`

Path to file(s) to be shut down. If more than one file, use a colon separator.

`-vendor name`

If you specify a vendor name, only that vendor daemon is shut down, and `lmgrd` is not shut down.

`-q` Do not issue the `Are you sure?` prompt.

You should restrict the execution of `lmdown` to license administrators, by starting `lmgrd` with the `-p -2` switch, as shutting down the server causes loss of licenses.

To disable `lmdown`, the license administrator can use `lmgrd -x lmdown`.

To stop and restart a single vendor daemon, use `lmdown -vendor name`, then `lmreread -vendor name`.

**lmhostid**

`lmhostid`

This program returns the correct host ID on any computer supported by FlexLM.

**lmremove**

`lmremove [-c license_file_list] feature user host display`

This utility allows you to remove a single user license for a specific feature. For example, when a user is running the software and the host crashes, the user license is sometimes left checked out and unavailable to other users. `lmremove` frees the license and makes it available to other users.

`-c license_file_name`

The full pathname of the license file to be used. If this is omitted the LM\_LICENSE\_FILE environment variable is used instead.

*feature* The name of the feature the user has checked out.

*user* The name of the user.

*host* The name of the host the user was logged into.

*display* The name of the display where the user was working.

You can obtain the `user`, `host`, and `display` information from the output of `lmstat -a`.

If the application is active when its license is removed by `lmremove`, it checks out the license again at the next application heartbeat.

**lmreread**

`lmreread [-vendor name] [-c license_file_list]`

This utility causes the license daemon to reread the license file, and start any new vendor daemons that have been added. All the existing daemons are signalled to reread the license file to check for any changes in their licensing information.

`-vendor name`

If you specify a vendor name, only that vendor daemon rereads the license file. If the vendor daemon is not running, `lmgrd` starts it.

To disable `lmreread`, the license administrator can use `lmgrd -x lmreread`.

`lmreread` does not cause server host names or port numbers to be reread from the license file. To make any changes to those items effective, you must restart `lmgrd`.

To stop and restart a single vendor daemon, use `lmdown -vendor name`, then `lmreread -vendor name`.

## lmstat

```
lmstat [-a] [-A] [-c license_file_list] [-f [feature]] [-i
[feature]] [-s [server]] [-S [daemon]] [-t value]
```

This utility helps you to monitor the status of all network licensing activities, including:

- which daemons are running
- users of individual features
- users of features served by specific daemons.

The optional arguments are:

```
-a          Displays all information.
-A          Lists all active licenses.
-c license_file_list
            Uses all the license files listed.
-f [feature]
            List users of a specific feature.
-i [feature]
            Print information about the named feature, or all features if feature is
            omitted.
-s [server]
            Display status of server node(s).
-S [daemon]
            List all users and features of a specific daemon.
-t value   Set the lmstat time-out to value.
```

## lmver

```
lmver [filename]
```

This utility reports the FlexLM version of a specific library or binary file.

## A.10 Frequently asked questions about licensing

**Q** *Why can I not find the LMHOSTID program?*

**A** You have to run the `makelinks.sh` script that is in the directory containing the FlexLM software. This script creates a series of links to the `lmutil` program, one of which is for `lmhostid`.

**Q** *How does an application find its license file?*

**A** An application and the license server software itself looks in the following places for license files:

```
$ARMLMD_LICENSE_FILE
$LM_LICENSE_FILE
/opt/arm/licenses
/usr/local/flexlm/license.dat
```

The `$ARMLMD_LICENSE_FILE` and `$LM_LICENSE_FILE` environment variables can each contain multiple license file names, separated by colons. In addition to full pathnames to files, they can hold directory names. If the license software finds a directory name it will search that directory looking for files that end with `.lic` and treat all such files as license files.

`/opt/arm/licenses` is the default location that ARM applications search for their license file.

**Q** *Do I need to have the license file on my client machine?*

**A** Sometimes. You need to have the license file on your client machines only when you are using the three-license server option.

In this situation you need to point the `ARMLMD_LICENSE_FILE` environment variable at the local copy of the license file. Ensure that the hostnames and TCP port numbers in the local license file are the same as in the license server copies.

On a single-license server you can normally set `ARMLMD_LICENSE_FILE` to contain the hostname of the server.



# Index

## Symbols

@ and ^ indicators 2-6

## Numerics

16/32-bit assembly code 2-2

## A

accessing

host peripherals 1-4

online help 1-2

address, realtime 1-4

ADU (ARM Debugger for UNIX)

closing down 3-15

desktop 3-1

starting up 3-14

startup configuration 5-7

agent, debug 2-2

analysis of processor time 2-7, 4-10

Angel 1-4, 2-2

configuring 5-12

Angel Debug Monitor (ADM) 1-5

Angel Debug Protocol (ADP) 1-5

ANSI C library 1-4

arguments, command line 4-8

ARM core 1-4

ARM processors 1-4

ARM Software Development Toolkit iii

ARM/Thumb code 2-2

armprof 2-7

ARMulator 1-4, 2-2

configuring 5-9

ASIC 1-4

assembly code, 16/32-bit 2-2

## B

backtrace window 2-4, 3-7

benchmarking 1-4

book

commenting on vi

structure iv

breakpoints

complex 4-2

data-dependent 1-4

removing 3-19

setting, editing and deleting 2-3, 4-2

simple 3-18

simple and complex 2-3, 3-18

viewing 3-19

breakpoints window 2-3, 3-7

## C

C programs 1-4

changing

variables 3-21

characters, special 2-5

closing debugger 3-15

code

ARM/Thumb 2-2, 3-27

disassembled 2-4

efficiency of 2-7

command line arguments 4-8

command line debugger instructions 4-9

command window 3-6

- comments vi
- complex
  - breakpoint 2-3, 4-2
  - watchpoint 2-3, 4-3
- concepts and terminology 2-1
- configuring
  - ADU startup 5-7
  - ARMulator 5-9
  - debugger 5-3
  - EmbeddedICE 5-14
  - Remote\_A 5-12
  - target environments 5-2
- console window 3-5
- conventions used v
- copying disk file to memory 4-6

## D

- debug agent 2-2
- debugger
  - closing down 3-15
  - command line instructions 4-9
  - commenting on vi
  - configuring 5-3
  - how to use 3-14
  - introduction to 1-1
  - starting up 3-14
  - strategy for using 1-3
- debugger internals window 3-8
- deleting
  - breakpoints 2-3, 3-19, 4-2
  - watchpoints 3-20
- desktop 3-1
- directory 2-5
- disassembly
  - mode 3-27
  - window 2-4, 3-10
- disk copy of memory area 4-6
- display formats 3-25
- displaying
  - image information 3-23

## E

- editing breakpoints 2-3, 4-2
- EmbeddedICE 1-4, 2-2
  - configuring 5-14
- emulator 1-4
- environment configuration 5-2
- examining
  - memory 3-22, 3-26
  - search paths 3-23
  - source files 3-24
  - variables 3-24
- executing
  - ADU 3-14
  - an image 3-16
- execution
  - starting 3-16
  - stopping 3-17, 3-18
- execution window 3-5
- exiting debugger 3-15
- expression window 3-10
- expressions, regular 2-5

## F

- feedback vi
- file location 2-5
- files, memory map 5-5
- flash download 4-11
- FlexLM
  - management of multiple licenses A-12
  - versions A-12
- formatting displayed variables 3-25
- function call, stepping to 3-17
- function names window 2-6, 3-11
- function, stepping out of 3-17

## G

- getting started 3-14
- global memory map file 5-5
- globals window 3-11

## H

- help, online 1-2
- high-level symbols 2-6
- host peripherals, accessing 1-4

## I

- image
  - displaying information 3-23
  - executing 3-16
  - loading 3-15
  - path 2-5
  - reloading 3-16
  - stepping through 3-17
- indicators, @ and ^ 2-6
- indirection 3-26
- instructions, command line 4-9
- introduction to debugging 1-1

## J

- JTAG 1-4

## K

- key to conventions v

## L

- library, C 1-4
- license file
  - customizing A-9
  - obtaining A-4
  - typical A-9
- licenses, multiple A-12
- lmchecksum utility A-14
- lmdiag utility A-15
- lmdown utility A-15
- lmhostid utility A-16
- lmremove utility A-16
- lmreread utility A-16
- lmstat utility A-17
- lmver utility A-17
- loading an image 3-15

- local memory map file 5-6
- locals window 3-11
- location of files 2-5
- low level symbols 2-6
- low level symbols window 3-12

## M

- matching strings 2-5
- memory
  - examining 3-22, 3-26
  - flash 4-11
  - map files 5-5
  - restoring from disk 4-6
  - saving to disk 4-6
- memory window 3-12
- menu bar 3-3
- menus 3-4
- mode, disassembly 3-27
- modifying
  - variables 3-21

## N

- next line, stepping to 3-17

## O

- online help, accessing 1-2
- operating the debugger 3-14

## P

- paths, search 2-5, 3-23
- peripherals, accessing 1-4
- processor time analysis 4-10
- product, comments on vi
- profiling 2-7, 4-10
- program image
  - displaying information 3-23
  - executing 3-16
  - loading 3-15
  - reloading 3-16
  - stepping through 3-17

- programs, C 1-4
- properties of variables 3-26

## Q

- quitting debugger 3-15

## R

- RDI (Remote Debug Interface) 2-2
- RDI log window 3-13, 3-28
- realtime address 1-4
- registers
  - halting if changed 2-3
- registers window 3-12
- regular expressions 2-5
- reloading an image 3-16
- remote debug information 3-28
- Remote\_A 1-5
  - configuring 5-12
- Remote\_D 1-5
- removing
  - breakpoints 3-19
  - watchpoints 3-20
- restoring memory from disk 4-6
- run to cursor 3-17

## S

- saving memory to disk 4-6
- search paths 2-5
  - viewing 3-23
- search paths window 2-5, 3-13
- setting
  - breakpoints 2-3, 4-2
  - simple breakpoint 3-18
  - simple watchpoint 3-20
- simple
  - breakpoint 2-3, 3-18
  - watchpoint 2-3, 3-20
- single stepping 1-4
- Software Development Toolkit iii
- software, ARM-targeted 1-4
- source file window 3-13

- source files
  - examining 3-24
- source files list window 3-13
- special characters 2-5
- specifying strings 2-5
- starting ADU 3-14
- startup configuration 5-7
- status bar 3-3
- stepping through an image 3-17
- stopping ADU 3-15
- stopping execution 3-17
- strategy for debugging 1-3
- strings, specifying and matching 2-5
- structure of book iv
- subdirectory 2-5
- suggestions vi
- symbols, high- and low-level 2-6

## T

- target environment configuration 5-2
- terminology and concepts 2-1
- Thumb code 2-2
- time analysis 2-7, 4-10
- toolbar 3-3
- toolkit for software development iii
- typographical conventions v

## U

- using the debugger 3-14

## V

- variables
  - halt if changed 2-3
  - properties of 3-26
  - viewing 3-24
  - viewing and changing 3-21
- view menu 3-4, 3-23
- viewing
  - breakpoints 3-19
  - memory 3-22, 3-26
  - search paths 3-23

- source files 3-24
- variables 3-21, 3-24
- watchpoints 3-20

## W

- watchpoints
  - complex 4-3
  - removing 3-20
  - simple 3-20
  - simple and complex 2-3, 3-18
  - viewing 3-20
- watchpoints window 3-13
- window
  - backtrace 2-4, 3-7
  - breakpoints 2-3, 3-7
  - command 3-6
  - console 3-5
  - debugger internals 3-8
  - disassembly 2-4, 3-10
  - execution 3-5
  - expression 3-10
  - function names 2-6, 3-11
  - globals 3-11
  - locals 3-11
  - low level symbols 3-12
  - memory 3-12
  - RDI log 3-13, 3-28
  - registers 3-12
  - search paths 2-5, 3-13
  - source file 3-13
  - source files list 3-13
  - watchpoints 3-13
- windows
  - available on desktop 3-1