ARM® Developer Suite

Version 1.1

Debug Target Guide

Copyright © 1999, 2000 ARM Limited. All rights reserved.

ARM DUI0058C

ARM Developer Suite
Debug Target Guide

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

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

Proprietary Notice

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

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

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

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

Contents
Debug Target Guide

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Preface

This preface introduces the ARM debug targets and their reference documentation. It contains the following sections:

- *About this book* on page Preface-vi
- *Feedback* on page Preface-x.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

About this book

This book provides reference information for the *ARM Developer Suite* (ADS). It describes:

- ARMulator®, the ARM simulator
- Angel™, the ARM debug monitor
- Semihosting SWIs, the means for your ARM programs to access facilities on your host computer.

Intended audience

This book is written for all developers who are using the ARM debuggers, `armsd`, AXD, ADU or ADW. It assumes that you are an experienced software developer, and that you are familiar with the ARM development tools as described in *Getting Started*.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the material in this book, and a summary description of the range of ARM debug targets.

Chapter 2 *ARMulator Basics*

Read this chapter for a description of ARMulator, the ARM instruction set simulator.

Chapter 3 *Writing ARMulator Models*

Read this chapter for help in writing your own extensions and modifications to ARMulator.

Chapter 4 *ARMulator Reference*

This chapter provides further details to help you use ARMulator.

Chapter 5 *Semihosting*

Read this chapter for information about how to access facilities on the host computer from your ARM programs.

Typographical conventions

The following typographical conventions are used in this book:

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

Further reading

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

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

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

ARM publications

This book contains information that is specific to the versions of ARMulator, Angel and the semihosting SWIs supplied with the *ARM Developer Suite* (ADS). Refer to the following books in the ADS document suite for information on other components of ADS:

- *ADS Installation and License Management Guide* (ARM DUI 0139)
- *Getting Started* (ARM DUI 0064)
- *ADS Assembler Guide* (ARM DUI 0068)
- *ADS Compiler, Linker, and Utilities Guide* (ARM DUI 0067)
- *CodeWarrior IDE Guide* (ARM DUI 0065)
- *ADS Debuggers Guide* (ARM DUI 0066)
- *ADS Developer Guide* (ARM DUI 0056)
- *ARM Applications Library Programmer's Guide* (ARM DUI 0081).

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

- *ARM Architecture Reference Manual* (ARM DDI 0100). This is supplied in DynaText format as part of the online books, and in PDF format in `install_directory\PDF\ARM-DDI0100B_armarm.pdf`.
- *ARM ELF specification* (SWS ESPC 0003). This is supplied in PDF format in `install_directory\PDF\specs\ARMELF.pdf`.
- *TIS DWARF 2 specification*. This is supplied in PDF format in `install_directory\PDF\specs\TIS-DWARF2.pdf`.
- *ARM/Thumb® Procedure Call Specification*. This is supplied in PDF format in `install_directory\PDF\specs\ATPCS.pdf`.

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

- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Feedback

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

Feedback on the ARM Developer Suite

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

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small stand-alone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you have any problems with this book, please send email to `errata@arm.com` giving:

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

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the debug support facilities provided in the ADS version 1.1. It contains the following section:

- *Debug target overview* on page 1-2.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

1.1 Debug target overview

You can debug your prototype software using any of the debuggers described in *ADS Debuggers Guide*. The debugger runs on your *host computer*. It is connected to a *target system* that your prototype software runs on.

Your target system can be any one of:

- a software simulator, simulating ARM hardware
- an ARM evaluation or development board
- a third-party ARM-based development board
- ARM-based hardware of your own design.

1.1.1 ARMulator

A software simulator, ARMulator, is supplied with ADS. ARMulator runs on the same host computer as the debugger. It includes facilities for communication with the debugger.

ARMulator is an instruction set simulator. It simulates the instruction sets and architecture of ARM processors, together with a memory system and peripherals. You can extend it to simulate other peripherals and custom memory systems (see Chapter 3 *Writing ARMulator Models*).

You can use ARMulator for software development and for benchmarking ARM-targeted software. It models the instruction set and counts cycles.

This book is mainly concerned with the ARMulator.

1.1.2 Hardware targets

You can use one of three different arrangements for a debugger to communicate with a hardware target system:

- You can run a debug monitor, such as Angel or RealMonitor, on the target system, in addition to your prototype code. The debug monitor handles communication with the debugger.

- If your target system has EmbeddedICE® logic, you can set:
  - breakpoints in your code
  - watchpoints in memory.

  Execution halts at breakpoints, or when watchpoints are accessed. You can then examine the state of your system, alter it, and restart it. In this way you can avoid having any code other than your prototype software running on your target system.

- If your target system has an Embedded Trace Macrocell, you can examine the operation of your system while it is running. This can be done with only your prototype software running on your target system.

For details see the documentation accompanying the hardware.

1.1.3 Semihosting

You can use the I/O facilities of the host computer, instead of providing the facilities on your target system. This is called *semihosting* (see Chapter 5 *Semihosting*).

To access these, use semihosting *Software Interrupts* (SWIs). Any of the following intercept semihosting SWIs and request service from the host computer:

- ARMulator
- your debug monitor
- Multi-ICE®.

Chapter 2
ARMulator Basics

This chapter describes ARMulator, a collection of programs that provide software simulation of ARM processors. It contains the following sections:

2.1 About ARMulator

ARMulator is an instruction set simulator. It simulates the instruction sets and architecture of various ARM processors. To run software on ARMulator, you must access it either through the ARM symbolic debugger, `armsd`, or through the ARM GUI debuggers, AXD, ADU, or ADW. See *ADS Debuggers Guide* for details.

ARMulator is suited to software development and benchmarking ARM-targeted software. It models the instruction set and counts cycles.

ARMulator supports a full ANSI C library to allow complete C programs to run on the simulated system. Refer to the library chapter in *ADS Compiler, Linker, and Utilities Guide* for more information on C and C++ library support. See also Chapter 5 *Semihosting* for information on the C library semihosting SWIs supported by ARMulator.

2.2 ARMulator components

ARMulator consists of a series of modules, implemented as *Dynamic Link Libraries* (`.dll` files) for Windows, or as *Shared Objects* (`.so` files for UNIX, `.sl` files for HPUX).

The main modules are:

- a model of the ARM processor core
- a model of the memory used by the processor.

There are alternative predefined modules for each of these parts. You can select the combination of processor and memory model you want to use.

One of the predefined memory models, `armmap`, allows you to specify a simulated memory system in detail.

In addition there are predefined modules which you can use to:

- model additional hardware, such as a coprocessor or peripherals
- model pre-installed software, such as a C library, semihosting SWI handler, or an operating system
- provide debugging or benchmarking information (see *Tracer* on page 2-5 and *Profiler* on page 2-11).

You can use different combinations of predefined modules, and different memory maps (see *Configuring ARMulator* on page 2-4 and *Memory model with memory map* on page 2-19).

You can write your own modules, or edit copies of the predefined ones, if the modules provided do not meet your requirements. For example:

- to model a different peripheral, coprocessor, or operating system
- to model a different memory system
- to provide additional debugging or benchmarking information.

The source code of some modules is supplied. You can use these as examples to help you write your own modules (see Chapter 3 *Writing ARMulator Models*).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.2.1 Configuring ARMulator

You can configure some of the details of ARMulator from `armsd`, or from your GUI debugger (see *ADS Debuggers Guide*). The current configurations are announced in the debugger startup banner.

To make other configuration adjustments you must edit ARMulator files directly. This usually involves editing `.ami` files. Three `.ami` files are supplied with ADS:

- `default.ami`
- `peripherals.ami`
- `example1.ami`.

These files are in *`install_directory`*`\Bin`. If you write any ARMulator models of your own, you may produce additional `.ami` files to allow your models to be configured.

When ARMulator is started by a debugger, it reads all the `.ami` files it finds on any of the paths it finds in the environment variable `armconf`. This is initially set up to point to *`install_directory`*`\Bin`.

The following sections describe each of the predefined modules in turn, and how they can be configured.

2.3 Tracer

You can use Tracer to trace instructions, memory accesses, and events. The configuration file `peripherals.ami` controls what is traced (see *ARMulator configuration files* on page 4-57).

This section contains the following subsections:

- *Debugger support for tracing* on page 2-5
- *Interpreting trace file output* on page 2-6
- *Configuring Tracer* on page 2-9.

2.3.1 Debugger support for tracing

There is no direct debugger support for tracing. Instead, Tracer uses bit 4 of the RDI logging level (`$rdi_log`) variable to enable or disable tracing.

Using AXD

Select **System Views** → **Debugger Internals** → **Internal Variables**, and then double-click on the `$rdi_log` value to edit it:

- To enable tracing, set `$rdi_log` to `0x00000010`.
- To disable tracing, set `$rdi_log` to `0x00000000`.

Using ADU or ADW

Select **Set RDI Log Level** from the **Options** menu:

- To enable tracing, set the RDI log level to 16.
- To disable tracing, set the RDI log level to 0.

Using armsd

Enter the following at the command prompt:

- To enable tracing under armsd, type `$rdi_log=16`.
- To disable tracing, type `$rdi_log=0`.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.3.2 Interpreting trace file output

This section describes how you interpret the output from Tracer.

Example of a trace file

The following example shows part of a trace file:

```
Date: Fri Jul 16 13:29:16 1999
Source: Armul
Options: Trace Instructions  (Disassemble)  Trace Memory Cycles
MNR4O__ 00008008 EB00000C
MSR4O__ 0000800C EB00001B
MSR4O__ 00008010 EF000011
IT 00008008 eb00000c BL      0x8040
MNR4O__ 00008040 E1A00000
MSR4O__ 00008044 E04EC00F
MSR4O__ 00008048 E08FC00C
IT 00008040 e1a00000 NOP
MSR4O__ 0000804C E99C000F
IT 00008044 e04ec00f SUB     r12,r14,pc
MSR4O__ 00008050 E24CC010
IT 00008048 e08fc00c ADD     r12,pc,r12
E 00000020 00000000 10005
MNR4O__ 00000020 E1A00000
IT 00000018 eb00000a BL      0x48
E 00000048 00000000 10005
MNR4O__ 00000048 E10F0000
E 0000004C 00000000 10005
MSR4O__ 0000004C E1A00000
```

In a trace file, there are three types of line:

- trace memory entries (M lines)
- trace instruction entries (I lines)
- trace event entries (E lines).

These are described in the following sections.

Trace memory (M lines)

Trace memory (M) lines have the following format for general memory accesses:

```
M<type><rw><size>[O][L][S] <address> <data>
```

where:

`<type>`	indicates the cycle type:
	`S` sequential

	`N`	nonsequential.
`<rw>`	indicates either a read or a write operation:	
	`R`	read
	`W`	write.
`<size>`	indicates the size of the memory access:	
	`4`	word (32 bits)
	`2`	halfword (16 bits)
	`1`	byte (8 bits).
`O`	indicates an opcode fetch (instruction fetch).	
`L`	indicates a locked access.	
`S`	indicates a speculative instruction fetch.	
`<address>`	gives the address in hexadecimal format, for example `00008008`.	
`<data>`	can show one of the following:	
	`value`	gives the read/written value, for example `EB00000C`
	`(wait)`	indicates **nWAIT** was LOW to insert a wait state
	`(abort)`	indicates **ABORT** was HIGH to abort the access.

Trace memory lines can also have any of the following formats:

- `MI`
  for idle cycles
- `MC`
  for coprocessor cycles
- `MIO`
  for idle cycles on the instruction bus of Harvard architecture processors such as ARM9TDMI™.

Trace instructions (I lines)

The format of the trace instruction (I) lines is as follows:

```
[ IT | IS ] <instr_addr> <opcode> [<disassembly>]
```

For example:

```
IT 00008044 e04ec00f SUB      r12,r14,pc
```

where:

`IT`	indicates that the instruction was taken.
`IS`	indicates that the instruction was skipped (all ARM instructions are conditional).
`<instr_addr>`	shows the address of the instruction in hexadecimal format, for example `00008044`.
`<opcode>`	gives the opcode in hexadecimal format, for example `e04ec00f`.
`<disassembly>`	gives the disassembly (uppercase if the instruction is taken), for example, `SUB r12,r14,pc`. This is optional and is enabled by setting `Disassemble=True` in `peripherals.ami`.

Branches and branches with link in Thumb code appear as two entries, with the first marked:

```
1st instr of BL pair.
```

Events (E lines)

The format of the event (E) lines is as follows:

```
E <word1> <word2> <event_number>
```

For example:

```
E 00000048 00000000 10005
```

where:

`<word1>`	gives the first of a pair of words, such as, the pc value.
`<word2>`	gives the second of a pair of words, such as, the aborting address.
`<event_number>`	gives an event number, for example `0x10005`. This is MMU `Event_ITLBWalk`. Events are described in *Events* on page 4-27.

2.3.3 Configuring Tracer

Tracer has its own section in the ARMulator peripherals configuration file (`peripherals.ami`):

```
{ Default_Tracer=Tracer
;; Output options - can be plaintext to file, binary to file or to RDI log
;; window. (Checked in the order RDILog, File, BinFile.)
RDILog=False
File=armul.trc
BinFile=armul.trc
;; Tracer options - what to trace
TraceInstructions=True
TraceRegisters=False
TraceMemory=False
TraceIdle=True
TraceNonAccounted=True
TraceEvents=False
;; Where to trace memory - if not set, it will trace at the core.
TraceBus=True
;; Flags - disassemble instructions; start up with tracing enabled;
Disassemble=True
StartOn=False
}
```

where:

`RDILog`	instructs Tracer to output to the RDI log window (in the GUI debuggers) or the console (under `armsd`).
`File`	defines the file where the trace information is written. Alternatively, you can use `BinFile` to store data in a binary format.

The other options control what is being traced:

`TraceInstructions`	traces instructions.
`TraceRegisters`	traces registers.
`TraceMemory`	traces real memory accesses.
`TraceIdle`	traces idle cycles.
`TraceNonAccounted`	traces unaccounted RDI accesses to memory. That is, those accesses made by the debugger.
`TraceEvents`	traces events. For more information, see *Tracing events* below.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

`TraceBus`	controls the trace data source. This is one of:
	`TRUE` Bus (between processor and memory)
	`FALSE` Core (between core and cache, if present).
`Disassemble`	disassembles instructions. Simulation is much slower if you enable disassembly.
`StartOn`	instructs ARMulator to trace as soon as execution begins.

Other tracing controls

You can also control tracing using:

`Range=`*`low address`*`,`*`high address`*

tracing is carried out only within the specified address range.

`Sample=`*`n`* only every *`n`*th trace entry is sent to the trace file.

Tracing events

When tracing events, you can select the events to be traced using:

`EventMask=`*`mask`*`,`*`value`*

only those events whose number when masked (bitwise-AND) with *`mask`* equals *`value`* are traced.

`Event=`*`number`* only *`number`* is traced. (This is equivalent to `EventMask=0xFFFFFFFF,`*`number`*.)

For example, the following traces only MMU/cache events:

```
EventMask=0xFFFF0000,0x00010000
```

See *Events* on page 4-27 for more information.

2.4 Profiler

Profiler is controlled by the debugger. For more details on Profiler, see Chapter 4 *ARMulator Reference*.

In addition to profiling program execution time, Profiler allows you to use the profiling mechanism to profile events, such as cache misses.

2.4.1 Configuring Profiler

Profiler has its own section in the ARMulator peripherals configuration file (`peripherals.ami`):

```
{ Default_Profiler=Profiler
;; For example - to profile the PC value when cache misses happen, set:
;Type=Event
;Event=0x00010001
;EventWord=pc

;;Alternatives for Type are
;;  Event, Cycle, Microsecond.
;;If type is Event then alternatives for EventWord are
;;  Word1,Word2,PC.
}
```

Every line in this section is a comment, so the ARMulator will perform its default profiling. The default is to take profiling samples at intervals of 100 microseconds. Refer to *ADS Debuggers Guide* for further information.

If this section is uncommented, data cache misses will be profiled. See *Events* on page 4-27 for more information.

The `Type` entry controls how the profiling interval is interpreted:

`Type=Microsecond`
: instructs Profiler to take samples every *n* microseconds. This is the default.

`Type=Cycle`
: instructs Profiler to take samples every *n* instructions, and record the number of memory cycles since the last sample.

`Type=Event`
: instructs Profiler to ignore the profiling interval. Instead, it profiles relevant events, see *Events* on page 4-27.

`EventMask=`*`mask,value`* is also allowed (see *Tracer* on page 2-5).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.5 Pagetable module

This section contains the following subsections:

2.5.1 Overview of the pagetable module

On models of ARM architecture v4 processors with a *memory management unit* (MMU), the pagetable module sets up pagetables and initializes the MMU. On processors with a *protection unit* (PU), the pagetable module sets up the PU. To control whether to include the pagetable model, find the `Pagetables` tag in the ARMulator configuration file, `default.ami`, and alter it as appropriate:

```
{Pagetables=Default_Pagetables
}
```

or

```
{ Pagetables=No_Pagetables
}
```

The `Pagetables` section in `peripherals.ami` controls the contents of the pagetables, and the configuration of the caches and MMU or PU. To locate the `Pagetables` section, find this line:

```
{PTI=Pagetables
```

For full details of the flags, control register and pagetables described in this section, see the datasheet or technical reference manuals for the processor you are simulating.

Note

This module allows you to benchmark or debug code. You must write ARM code to set up the MMU or PU for a real system.

2.5.2 Controlling the MMU or PU and cache

The first set of flags enables or disables features of the caches and MMU or PU:

```
MMU=Yes
AlignFaults=No
Cache=Yes
WriteBuffer=Yes
Prog32=Yes
Data32=Yes
LateAbort=Yes
BigEnd=No
BranchPredict=Yes
ICache=Yes
HighExceptionVectors=No
FastBus=No
```

Each flag corresponds to a bit in the system control register 1.

Some flags only apply to certain processors. For example, `BranchPredict` only applies to the ARM810™, and `ICache` to the SA™-110 and ARM940T™ processors. These flags are ignored by other processor models.

The `FastBus` flag is used by the ARM940T™. If your system uses Fast Bus Mode, set `FastBus=Yes` for benchmarking. If you do not set `FastBus`, ARMulator assumes that the memory is synchronous with the core.

The MMU flag is also used in processors with a PU.

2.5.3 Controlling registers 2 and 3

The following options apply only to processors with an MMU:

```
PageTableBase=0xA0000000
DAC=0x00000001
```

They control:

- the translation table base register (system control register 2)
- the domain access control register (system control register 3).

You must align the address in the translation table base register to a 16KB boundary.

2.5.4 Memory regions

The rest of the Pagetables configuration section defines a set of memory regions. Each region has its own set of properties.

By default, `peripherals.ami` contains a description of a single region covering the whole of the address space:

```
{ Region[0]
VirtualBase=0
PhysicalBase=0
Size=4GB
Cacheable=Yes
Bufferable=Yes
Updateable=Yes
Domain=0
AccessPermissions=3
Translate=Yes
}
```

You can add more regions following the same general form:

`Region[`*n*`]`	names the regions, starting with `Region[0]`. *n* is an integer.
`VirtualBase`	applies only to a processor with an MMU. It gives the address of the base of the region in the virtual address space of the processor. This address must be aligned to a 1MB boundary. It is mapped to `PhysicalBase` by the MMU.
`PhysicalBase`	gives the physical address of the base of the region. On a processor with an MMU, this address must be aligned to a 1MB boundary. On a processor with a PU it must be aligned to a boundary that is a multiple of the size of the region.
`Size`	specifies the size of this region. On a processor with an MMU `Size` must be a whole number of megabytes. On a processor with a PU, `Size` must be 4KB or a power-of-two multiple of 4KB.
`Cacheable`	specifies whether the region is to be marked as cacheable. If it is, reads from the region will be cached.
`Bufferable`	specifies whether the region is to be marked as bufferable. If it is, writes to the region will use the write buffer.
`Updateable`	applies only to the ARM610™ processor. It controls the `U` bit in the translation table entry.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

`Domain`	applies only on processors with an MMU. It specifies the domain field of the table entry.
`AccessPermissions`	specifies the access controls to the region. Refer to the processor datasheet for further information.
`Translate`	controls whether accesses to this region cause translation faults. Setting `Translate=No` for a region causes an abort to occur whenever the processor reads from or writes to that region.

2.5.5 Pagetable module and memory management units

Processors such as ARM710T™ and ARM920T™ have an MMU.

An MMU uses a set of page tables, stored in memory, to define memory regions. On reset, the pagetable module writes out a top-level page table to the address specified in the translation table base register. The table corresponds to the regions you define in the `Pagetables` section of `peripherals.ami`.

For example, the default configuration details, given in *Memory regions* on page 2-14, define the following page table:

- The entire address space, 4GB, is defined as a single region. This region is cacheable and bufferable. Virtual addresses are mapped directly to the same physical addresses over the whole address space.
- The translation table base register, register 2, is initialized to point to this page table in memory, at `0xA0000000`.
- The domain access control register, register 3, is initialized with value `0x00000001`. This sets the access to the region as *client*.
- The M, C and W bits of the control register, register 1, are configured to enable the MMU, cache, and write buffer. If the processor has separate instruction and data caches, the I bit configures the instruction cache enabled.

2.5.6 Pagetable module and protection units

Processors such as ARM740T™ and ARM940T™ have a PU.

A PU uses a set of protection regions. The base and size of each protection region is stored in registers in the PU. On reset, the page table module initializes the PU.

For example, the default configuration details given above define a single region, region 0. This region is marked as read/write, cacheable, and bufferable. It occupies the whole address range, 0 to 4GB.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

ARM740T PU

For an ARM740T, the PU is initialized as follows:

- The P, C, and W bits are set in the configuration register, register 1, to enable the protection unit, the cache and the write buffer.
- The cacheable register, register 2, is initialized to 1, marking region 0 as cacheable.
- The write buffer control register, register 3, is initialized to 1, marking region 0 as bufferable.
- The protection register, register 5, is initialized to 3, marking region 0 as read/write access.
- The protection region base and size register for region 0 is initialized to `0x3F`, marking the size of region 0 as 4GB and marking the region as enabled. The protection region base and size register for region 0 is part of register 6. Register 6 is actually a set of eight registers, each being the protection region base and size register for one region. See the datasheet for the processor for further details.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

ARM940T PU

For an ARM940T, the PU is initialized as follows:

- The P, D, W, and I bits are set in the configuration register, register 1, to enable the PU, the write buffer, the data cache and the instruction cache.
- Both the cacheable registers, register 2, are initialized to 1, marking region 0 as cacheable for the I and D caches. This is displayed in the debugger as `0x0101`, where:
  - the low byte (bits 0..7) represent the data cache cacheable register
  - the high byte (bits 8..15) represent the instruction cache cacheable register.
- The write buffer control register, register 3, is initialized to 1, marking region 0 as bufferable. This applies only to the data cache. The instruction cache is read only.
- Both the protection registers, register 5, are initialized to 3, marking region 0 as allowing full access for both instruction and data caches. This is displayed in the debugger as `0x00030003`, where:
  - the low halfword (bits 0..15) represent the data cache protection register
  - the high halfword (bits 16..31) represent the instruction cache protection register.

  The first register value shown is for region 0, the second for region 1 and so on.
- The protection region base and size register for region 0 is initialized to `0x3F`, marking the size of region 0 as 4GB and marking the region as enabled. The protection region base and size register for region 0 is part of register 6. Register 6 is really a set of sixteen registers, each being the protection region base and size register for one region. See the data sheet for the processor for further details.
- Register 7 is a control register. Reading from it is unpredictable. At startup the debugger shows a value of zero. It is not written to by the page table module.
- The programming lockdown registers, register 9, are both initialized to zero. The first register value shown in the debugger is for data lockdown control, the second is for instruction lockdown control.
- The test and debug register, register 15, is initialized to zero. Only bits 2 and 3 have any effect in ARMulator. These control whether the cache replacement algorithm is random or round-robin.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.6 Default memory model

The default memory model is a model of a zero-wait state memory system. The simulated memory size is not fixed. Host memory is allocated in chunks of 64KB each time a new region of memory is accessed. The memory size is limited by the host computer, but in theory all 4GB of the address space is available. The default memory model does not generate aborts.

The default memory model is used if you do not specify a mapfile in AXD, ADU, or ADW.

`armsd` looks in the current directory for a file called `armsd.map`. If it cannot find one, the default memory model is used.

Note

ARMulator does not load mapfiles itself. ARMulator receives mapfile information from the debugger that invokes it.

The default memory model routes memory accesses to memory-mapped peripheral models as appropriate. Routing is based on configuration details you provide in `peripherals.ami`, or another `.ami` file.

2.7 Memory model with memory map

This section contains the following subsections:

2.7.1 Overview of memory model with memory map

ARMmap is a memory model which you can configure yourself. You can specify the size, access width, access type and access speeds of individual memory blocks in the memory system in a memory map file (see *Map files* on page 4-53).

The debugger internal variables `$memstats` and `$statistics` give details of accesses of each cycle type, regions of memory accessed and time spent accessing each region.

The map memory model can generate aborts if you specify a memory region with access type as - (hyphen).

2.7.2 Clock frequency

You must specify a simulated clock frequency when using the map memory model. To configure the clock frequency:

- Under `armsd`, use the command-line option `-clock` *`clockspeed`*.
- Under the ADW or ADU, select the **Configure debugger** option from the **Options** menu. In the debugger configuration dialog, click on **Configure** to display the ARMulator configuration dialog. This contains a **Clock Speed** box that you can edit to the required frequency.
- Under AXD, select **Options** → **Configure Target** → **Configure**, enter the required clock speed, and then click the **Emulated** button.

For more information, refer to *ADS Debuggers Guide*.

The clock frequency is used to determine the number of wait states to be added to each memory access, as well as to calculate time from number of cycles. If you do not specify a clock speed, a value of 1MHz is used.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.7.3 Selecting the ARMmap memory model

Under `armsd`, the map memory model is automatically selected as the memory model to use whenever an `armsd.map` file exists in the directory where `armsd` is started.

Under the AXD, ADU, or ADW, the map memory model is automatically selected whenever a memory map file is specified. Specify map files using the **Memory Maps** tab of the ARMulator configuration dialog.

```
;; If there's a memory mapfile, use that.
#if MemConfigToLoad && MEMORY_MapFile
Default=MapFile
#endif
```

2.7.4 How the map memory model calculates wait states

The memory map file specifies access times in nanoseconds for nonsequential/sequential reads/writes to various regions of memory. By inserting wait states, the map memory model ensures that every access from the ARM processor takes at least that long.

The number of wait states inserted is the least number required to take the total access time over the number of nanoseconds specified in the memory map file. For example, with a clock speed of 33MHz (a period of 30ns), an access specified to take 70ns in a memory map file results in two wait states being inserted, to lengthen the access to 90ns.

This can lead to inefficiencies in your design. For example, if the access time were 60ns (only 14% faster) the model would insert only one wait state (33% quicker).

A mismatch between processor clock-speed and memory map file can sometimes lead to faster processor speeds having worse performance. For example, a 100MHz processor (10ns period) takes five wait states to access 60ns memory (a total access time of 60ns). At 110MHz, the map memory model must insert six wait states (a total access time of 63ns). So the 100MHz-processor system is faster than the 110MHz processor. (This does not apply to cached processors, where the 110MHz processor would be faster.)

Note

Access times specified in the memory map file must include propagation delays and memory controller decode time as well as the access time of the memory devices. For example, a map file specifies 80ns for 70ns RAM if there is a 10ns propagation delay.

2.7.5 Configuring the map memory model

You can configure the map memory model to model several different types of memory controller, by editing its entry in the `peripherals.ami` file:

```
{ Default_Mapfile=Mapfile
CountWaitStates=False
AMBABusCounts=False
;SpotICyles=True|False
SpotICyles=False
;ISTiming=Late|Early|Speculative
ISTiming=Late
}
```

Counting wait states

By default, the model is configured to count wait states in `$statistics`. You can disable this by setting `CountWaitStates=False` in `peripherals.ami`.

ARMulator cannot count wait states for systems based on ARM10™ or XScale processors.

Counting AMBA™ decode cycles

You can configure the model to insert an extra decode cycle for every nonsequential access from the processor. This models the decode cycle seen on some AMBA bus systems. Enable this by setting `AMBABusCounts=True` in `peripherals.ami`.

Merged I-S cycles

All ARM processors, particularly cached processors, can perform a nonsequential access as a pair of idle and sequential cycles, known as *merged I-S cycles*. By default, the model treats these cycles as a nonsequential access, inserting wait states on the S-cycle to lengthen it for the nonsequential access.

You can disable this by setting `SpotISCycles=False` in `peripherals.ami`. However, this is likely to result in exaggerated performance figures, particularly when modeling cached ARM processors.

The model can optimize merged I-S cycles using one of three strategies:

Speculative This models a system where the memory controller hardware speculatively decodes all addresses on idle cycles. The controller can use both the I- and S-cycles to perform the access. This results in one less wait state.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Early This starts the decode when the ARM declares that the next cycle is going to be an S-cycle, that is, half-way through the I-cycle. This can sometimes result in one fewer wait state. (Whether or not there are fewer wait states depends on the cycle time and the nonsequential access time for that region of memory.)

This is the default setting. You can change this by setting `ISTiming=Spec` or `ISTiming=Late` in `peripherals.ami`.

Late This does not start the decode until the S-cycle. In effect all S-cycles that follow an I-cycle are treated as if they are N-cycles.

Refer to the processor datasheet or reference manual for details of merged I-S cycles.

2.8 Peripheral models

ARMulator includes several peripheral models. This section gives basic user information about them. For more detailed information, refer to Chapter 4 *ARMulator Reference*.

This section contains the following subsections:

- *Configuring ARMulator to use the peripheral models*
- *Interrupt controller* on page 2-24
- *Timer* on page 2-24
- *Watchdog* on page 2-25
- *Stack tracker* on page 2-26
- *Tube* on page 2-26.

2.8.1 Configuring ARMulator to use the peripheral models

Enable or disable each peripheral model by changing the relevant entry in the `default.ami` file, for example:

```
{ WatchDog=No_watchdog
}
```

can be changed to:

```
{ Watchdog=Default_WatchDog
}
```

2.8.2 Semihosting

The semihosting SWI handler configuration is controlled by a section in `peripherals.ami`. It has the following items

```
{Default_Semihost=Semihost
; Demon is only needed for validation.
DEMON=False
ANGEL=TRUE
AngelSWIARM=0x123456
AngelSWIThumb=0xab
; And the default memory map
HeapBase=0x00000000
HeapLimit=0x07000000
StackBase=0x08000000
StackLimit=0x07000000
}
```

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.8.3 Interrupt controller

The interrupt controller configuration is controlled by a section in `peripherals.ami`. It has the following items:

```
{ Default_Intctrl=Intctrl
Range:Base=0x0a000000
WAITS=0
}
```

`Range:Base` specifies the area in memory into which the interrupt controller registers are mapped. For details of the interrupt controller registers, see *Interrupt controller* on page 4-67.

`WAITS` specifies the number of wait states that accessing the interrupt controller imposes on the processor. The maximum is 30.

2.8.4 Timer

The timer configuration is controlled by a section in `peripherals.ami`. It has the following items:

```
{Default_Timer=Timer
Range:Base=0x0a800000
;Frequency of clock to controller.
CLK=20000000
;; Interrupt controller source bits - 4 and 5 as standard
IntOne=4
IntTwo=5
WAITS=0
}
```

`Range:Base` specifies the area in memory into which the timer registers are mapped. For details of the interrupt controller registers, see *Timer* on page 4-69.

`CLK` is used to specify the clock rate of the peripheral. This is usually the same as the processor clock rate.

`IntOne` specifies the interrupt line connection to the interrupt controller for timer 1 interrupts. `IntTwo` specifies the interrupt line connection to the interrupt controller for timer 2 interrupts.

`WAITS` specifies the number of wait states that accessing the timer imposes on the processor. The maximum is 30.

2.8.5 Watchdog

Use Watchdog to prevent a failure in your program locking up your system. Watchdog resets ARMulator if your program fails to access it before a predetermined time.

Note

ARM do not supply a hardware watchdog timer. This is a generic model of a watchdog timer. It is supplied to help users model their system environment.

The Watchdog configuration is controlled by a section in `peripherals.ami`. It has the following items:

```
{Default_WatchDog=WatchDog
Range:Base=0xb0000000
KeyValue=0x12345678
WatchPeriod=0x80000
IRQPeriod=3000
IntNumber=16
StartOnReset=True
RunAfterBark=True
WAITS=0
}
```

`Range:Base` specifies the area in memory into which the watchdog registers are mapped.

This is a two-timer watchdog.

If `StartOnReset` is `True`, the first timer starts on reset. If `StartOnReset` is `False`, the first timer starts only when your program writes the configured key value to the `KeyValue` register.

The first timer generates an **IRQ** after `WatchPeriod` memory cycles, and starts the second timer. The second timer times out after `IRQPeriod` memory cycles, if your program has not written the configured key value to the `KeyValue` register. Configure `IRQPeriod` to a suitable value to allow your program to react to the **IRQ**.

If `RunAfterBark` is `True`, Watchdog halts ARMulator if the second timer times out. You can continue to execute, or debug.

If `RunAfterBark` is `False`, Watchdog resets ARMulator and halts.

`IntNumber` specifies the interrupt line number that Watchdog is attached to.

`WAITS` specifies the number of wait states that accessing the watchdog imposes on the processor. The maximum is 30.

2.8.6 Stack tracker

The stack tracker examines the contents of the stack pointer (r13) after each instruction. It keeps a record of the lowest value and from this it can work out the maximum size of the stack. ARMulator runs more slowly with stack tracking enabled.

The `StackUse` model continually monitors the stack pointer and reports the amount of stack used in $statistics. It must be configured with the stack's location.

The stack tracker is disabled by default. To enable the stack tracker, edit `peripherals.ami`:

1. Find the line:

   ```
   { Default_StackUse=NoStackuse
   ```

2. Change it to:

   ```
   { Default_StackUse=Stackuse
   ```

Before initialization the stack pointer can contain values outside the stack limits. You must configure the stack limits so that the stack tracker can ignore values outside them.

```
{ Default_StackUse=Stackuse
StackBase=0x80000000
StackLimit=0x70000000
}
```

`StackBase` is the address of the top of the stack. `StackLimit` is a lower limit for the stack. Changing these values does not reposition the stack in memory. To reposition the stack, you must reconfigure the debug monitor model.

2.8.7 Tube

The tube is a memory-mapped register. If you write a printable character to it, the character appears on the console.

You can change the address at which the Tube is mapped. This is controlled by an entry in `peripherals.ami`:

```
{Default_Tube=Tube
Range:Base=0x0d800020
}
```

Chapter 3
Writing ARMulator models

This chapter is intended to assist you in writing your own models to add to ARMulator. It contains the following sections:

3.1 The ARMulator extension kit

You can add extra models to ARMulator without altering the existing models. Each model is self-contained, and communicates with ARMulator through defined interfaces. The definition of these interfaces is in Chapter 4 *ARMulator Reference*.

The source code of some models is in:

`install_directory\ARMulate\ARMulext`

There are also header files in:

`install_directory\ARMulate\ARMulif`

Use these files as examples to help you write your own models. To help you choose suitable models to examine, this chapter includes a list of them with brief descriptions of what they do (see *Supplied models* on page 3-3).

3.1.1 Supplied models

ARMulator is supplied with source code for the following groups of models:

- *Basic models*
- *Peripheral models* on page 3-4

Basic models

`tracer.c`	The tracer module can trace instruction execution and events from within ARMulator (see *Tracer* on page 4-51). You can link your own tracing code onto the tracer module, allowing real-time tracing for example.
`profiler.c`	The profiler module provides the profiling function (see *Profiler* on page 2-11). This includes basic instruction sampling and more advanced use, such as profiling cache misses. It does this by providing an `UnkRDIInfoHandler` that handles the profiling requests from the debugger (see *Unknown RDI information handler* on page 4-33).
`pagetab.c`	On reset, this module sets up cache, PU or MMU and associated pagetables inside ARMulator (see *Pagetable module* on page 2-12).
`stackuse.c`	If enabled this model tracks the stack size. Stack usage is reported in the ARMulator memory statistics. You can set the stack upper and lower bounds in the `peripherals.ami` file.
`nothing.c`	This model does nothing. You can use this in the `peripherals.ami` file to disable models (see *Configuring ARMulator to disable a model* on page 3-10).
`semihost.c`	This model provides the semihosting SWIs described in Chapter 5 *Semihosting*.
`dcc.c`	This is a model of a debug communications coprocessor.
`mapfile.c`	This model allows you to specify the characteristics of a memory system. See *Map files* on page 4-53 for further information.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Peripheral models

`intc.c`	See *Interrupt controller* on page 2-24. `intc` is a model of the interrupt controller peripheral described in the *Reference Peripherals Specification* (RPS).
`timer.c`	See *Timer* on page 2-24. `timer` is a model of the RPS timer peripheral. Two timers are provided. `timer` must be used in conjunction with an interrupt controller, but not necessarily `intc`.
`millisec.c`	A simple millisecond timer.
`watchdog.c`	Watchdog. See *Watchdog* on page 2-25. `watchdog` is a generic watchdog model. It does not model any specific watchdog hardware, but provides generic watchdog functions.
`tube.c`	Tube. See *Tube* on page 2-26. `tube` is a simple debugging aid. It allows you to check that writes are taking place to a specified location in memory.

3.2 Writing a new peripheral model

This section contains the following subsections:

- *Using a sample model as a template*
- *Return values*
- *Initialization, finalization, and state macros* on page 3-6.

3.2.1 Using a sample model as a template

To write a new model, the best procedure is to copy one of the supplied models and then edit the copy. To do this:

1. Select which model is closest to the model you want to write. This might be, for example, Tracer.
2. Copy the source file, in this case `tracer.c`, with a new name such as `mymodel.c`.
3. Copy the make subdirectory, in this case `tracer.b`, with a corresponding new name, in this case `mymodel.b`.
4. Find `Makefile` in *`install_directory`*`\ARMulate\armulext\intelrel`. Load it into a text editor and change all instances of `tracer` to `mymodel`.

You can now edit MyModel.

3.2.2 Return values

A model must return one of the following states:

`PERIP_OK`	If the model is able to service the request.
`PERIP_BUSY`	If a memory access requires wait-states. A model must not return this state to a debugger access.
`PERIP_DABORT`	If a peripheral asserts the **DABORT** signal on the bus.
`PERIP_NODECODE`	If the model has been called with an address which belongs to it, but which has no meaning to it. The memory model handles the call as a memory access.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.2.3 Initialization, finalization, and state macros

To help you to write new ARMulator models, the following six macros are provided in `minperip.h`:

- `BEGIN_INIT()`
- `END_INIT()`
- `BEGIN_EXIT()`
- `END_EXIT()`
- `BEGIN_STATE_DECL()`
- `END_STATE_DECL()`.

Use the following to define an initialization function for your model:

```
BEGIN_INIT(your_model)
{
    /*
     * (your initialization code here)
    */
}
END_INIT(your_model)
```

Use the following to define a finalization function for your model:

```
BEGIN_EXIT(your_model)
{
    /*
     * (your finalization code here)
    */
}
END_EXIT(your_model)
```

The `BEGIN_INIT()` macro defines a structure to hold any private data used by your model, and the `END_EXIT()` macro frees it. Declare the data structure using:

```
BEGIN_STATE_DECL(your_model)
    /*
     * (your private data here)
    */
END_STATE_DECL(your_model)
```

3.3 Building a new model

To build your new model:

1. Change your current directory to:

   `install_directory``\ARMulate\armulext\mymodel.b\intelrel`.

2. Depending on your system:

   - for Windows, type:
     `nmake`
   - for UNIX, type:
     `make`.

3. On Windows, `mymodel.dll` appears in:

   `install_directory``\ARMulate\armulext\mymodel.b\intelrel`

   Move `mymodel.dll` to:

   `install_directory``\Bin`

   This is where ARMulator expects to find models.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.4 Configuring ARMulator to use a new model

ARMulator determines which models to use by reading the `.ami` and `.dsc` configuration files. See *ARMulator configuration files* on page 4-57.

Before a new model can be used by ARMulator, you must add a `.dsc` file for your model, and references to it must be added to the configuration files `default.ami` and `peripherals.ami`.

The procedures are described in the following subsections:

- *Adding a .dsc file* on page 3-8
- *Editing default.ami and peripherals.ami* on page 3-9.

3.4.1 Adding a .dsc file

Create a file called `MyModel.dsc` and place it in *`install_directory`*`\Bin`. It must contain the following:

```
;; ARMulator configuration file type 3
{ Peripherals
  {MyModel
    META_SORDI_DLL=MyModel
  }
  {
    No_MyModel=Nothing
  }
}
```

where:

`MyModel``.dll` is the filename of your model.

3.4.2 Editing default.ami and peripherals.ami

This description assumes that your model was based on Tracer:

1. Load the `default.ami` file into a text editor, and find the following lines:

   ```
   {Tracer=Default_Tracer
   }
   ```

2. Add the reference to your model:

   ```
   {Tracer=Default_Tracer
   }

   {MyModel=Default_MyModel
   }
   ```

3. Save your edited `default.ami` file.

4. Load the `peripherals.ami` file into a text editor, and find the Tracer section:

   ```
   { Default_Tracer=Tracer
   ;; Output options - can be plaintext to file, binary to file or to RDI log
   ;; window. (Checked in the order RDILog, File, BinFile.)
   .
   .
   .
   ;; Flags - disassemble instructions; start up with tracing enabled;
   Disassemble=True
   StartOn=False
   }
   ```

5. Using this as an example, add a configuration section for your model. Depending on how much your model differs from Tracer, it may be easiest to edit a copy of the Tracer section.

6. Save your edited `peripherals.ami` file.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.5 Configuring ARMulator to disable a model

You can disable a model by changing its entry in `peripherals.ami`. For example, to disable the Tube model:

1. Find the following lines in `peripherals.ami`:

   ```
   {Default_Tube=Tube
   Range:Base=0x0d800020
   }
   ```

2. Change them to read:

   ```
   {Default_Tube=No_Tube
   ;Range:Base=0x0d800020
   }
   ```

This uses the `nothing.c` model to override the `tube.c` model. Leaving the `Range:Base` line as a comment line makes it easier to re-enable the original model.

Chapter 4
ARMulator Reference

This chapter gives reference information about ARMulator. It contains the following sections:

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.1 ARMulator models

ARMulator comprises a collection of models that simulate ARM-based hardware. They enable you to benchmark, develop, and debug software before your hardware is available.

4.1.1 Configuring models through ToolConf

ARMulator models are configured through `ToolConf`. `ToolConf` is a database of tags and values that ARMulator reads from configuration files (`.dsc` and `.ami` files) during initialization (see *ToolConf* on page 4-60).

A number of functions are provided for looking up values from this database. The full set of functions is defined in `*install_directory*\ARMulate\clx\toolconf.h`. All the functions take an opaque handle called a `toolconf`.

4.2 Communicating with the core

During initialization, all the models receive a pointer to an `mdesc` structure of type `RDI_ModuleDesc *`. They copy this structure into their own state as a field called `coredesc`. This is passed as the first parameter to most *ARMulif* (ARMulator interface) functions. ARMulator exports these functions to enable models to access the ARMulator state through this handle.

The following functions provide read and write access to ARM registers:

- *ARMulif_GetReg* on page 4-5
- *ARMulif_SetReg* on page 4-5
- *ARMulif_GetPC and ARMulif_GetR15* on page 4-6
- *ARMulif_SetPC and ARMulif_SetR15* on page 4-6
- *ARMulif_GetCPSR* on page 4-7
- *ARMulif_SetCPSR* on page 4-7
- *ARMulif_GetSPSR* on page 4-8
- *ARMulif_SetSPSR* on page 4-8.

A model must pass a pointer to their coredesc structure when calling a function in ARMulif that calls the core.

The following functions provide convenient access to specific bits or fields in the CPSR:

- *ARMulif_ThumbBit* on page 4-8
- *ARMulif_GetMode* on page 4-9.

The following functions call the read and write methods for a coprocessor:

- *ARMulif_CPRead* on page 4-9
- *ARMulif_CPWrite* on page 4-10.

Note

It is not appropriate to access some parts of the state from certain parts of a model. For example, you must not set the contents of an ARM register from a memory access function, because the memory access function can be called during simulation of an instruction. In contrast, it is sometimes necessary to set the contents of ARM registers from a SWI handler function.

A number of the following functions take an **`unsigned`** mode parameter to specify the processor mode. The mode numbers are defined in `armdefs.h`, and are listed in Table 4-1 on page 4-4.

In addition, the special value `CURRENTMODE` is defined. This enables `ARMulif_GetReg()`, for example, to return registers of the current mode.

Table 4-1 Defined processor modes

`USER32MODE`	`ABORT32MODE`
`FIQ32MODE`	`UNDEF32MODE`
`IRQ32MODE`	`SYSTEM32MODE`
`SVC32MODE`	

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.2.1 ARMulif_GetReg

This function reads a register for a specified processor mode.

Syntax

```
ARMword ARMulif_GetReg(RDI_ModuleDesc *mdesc, ARMword mode, unsigned reg)
```

where:

mdesc	is the handle for the core.
mode	is the processor mode. Values for mode are defined in `armdefs.h` (seeTable 4-1 on page 4-4).
reg	is the register to read. Valid values are 0 to 14 for registers r0 to r14.

Return

The function returns the value in the given register for the specified mode.

4.2.2 ARMulif_SetReg

This function writes a register for a specified processor mode.

Syntax

```
void ARMulif_SetReg(RDI_ModuleDesc *mdesc, ARMword mode,
                    unsigned reg, ARMword value)
```

where:

mdesc	is the handle for the core.
mode	is the processor mode. Mode numbers are defined in `armdefs.h` (see Table 4-1 on page 4-4).
reg	is the register to write. Valid values are 0 to 14 for registers r0 to r14.
value	is the value to be written to register `reg` for the specified processor mode.

Usage

You can use this function to write to any of the general purpose registers r0 to r14, the PC, SPSR, or CPSR.

4.2.3 ARMulif_GetPC and ARMulif_GetR15

This function reads the pc. `ARMulif_GetPC` and `ARMulif_GetR15` are synonyms.

Syntax

`ARMword ARMulif_GetPC(RDI_ModuleDesc *mdesc)`

`ARMword ARMulif_GetR15(RDI_ModuleDesc *mdesc)`

where:

`mdesc`	is the handle for the core.

Return

This function returns the value of the pc.

4.2.4 ARMulif_SetPC and ARMulif_SetR15

This function writes a value to the pc. `ARMulif_SetPC` and `ARMulif_SetR15` are synonyms.

Syntax

`void ARMulif_SetPC(RDI_ModuleDesc *mdesc, ARMword value)`

`void ARMulif_SetR15(RDI_ModuleDesc *mdesc, ARMword value)`

where:

`mdesc`	is the handle for the core.
`value`	is the value to be written to the pc.

4.2.5 ARMulif_GetCPSR

This function reads the CPSR.

Syntax

`ARMword ARMulif_GetCPSR(RDI_ModuleDesc *mdesc)`

where:

mdesc	is the handle for the core.

Return

The function returns the value of the CPSR.

4.2.6 ARMulif_SetCPSR

This function writes a value to the CPSR.

Syntax

`void ARMulif_SetCPSR(RDI_ModuleDesc *mdesc, ARMword value)`

where:

mdesc	is the handle for the core.
value	is the value to be written to the CPSR.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.2.7 ARMulif_GetSPSR

This function reads the SPSR for a specified processor mode.

Syntax

```
ARMword ARMulif_GetSPSR(RDI_ModuleDesc *mdesc, ARMword mode)
```

where:

mdesc	is the handle for the core.
mode	is the processor mode for the SPSR you want to read.

4.2.8 ARMulif_SetSPSR

This function writes a value to the SPSR for a specified processor mode.

Syntax

```
void ARMulif_SetSPSR(RDI_ModuleDesc *mdesc, ARMword mode, ARMword value)
```

where:

mdesc	is the handle for the core.
mode	is the processor mode for the SPSR you want to write.
value	is the value to be written to the SPSR for the specified mode.

4.2.9 ARMulif_ThumbBit

This function returns 1 if the core is in Thumb state, 0 if the core is in ARM state.

Syntax

```
unsigned ARMulif_ThumbBit(RDI_ModuleDesc *mdesc)
```

where:

mdesc	is the handle for the core.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.2.10 ARMulif_GetMode

This function reads the current processor mode.

Syntax

```
unsigned ARMulif_GetMode(RDI_ModuleDesc *mdesc)
```

where:

mdesc	is the handle for the core.

4.2.11 ARMulif_CPRead

This function calls the `read` method for a coprocessor. It also intercepts calls to read the FPE emulated registers.

Syntax

```
int ARMulif_CPRead(RDI_ModuleDesc *mdesc, unsigned cpnum,
                   unsigned reg, ARMword *data)
```

where:

mdesc	is the handle for the core.
cpnum	is the number of the coprocessor.
reg	is the number of the coprocessor register to read from, as indexed by `CR`*n* in an `LDC` or `STC` instruction.
data	is a pointer for the data read from the coprocessor register. The number of words transferred, and the order of the words, is coprocessor dependent.

Return

The function must return:

- `ARMul_DONE`, if the register can be read
- `ARMul_CANT`, if the register cannot be read.

4.2.12 ARMulif_CPWrite

This function calls the `write` method for a coprocessor. It also intercepts calls to write the FPE emulated registers.

Syntax

```
int ARMulif_CPWrite(RDI_ModuleDesc *mdesc, unsigned cpnum,
                    unsigned reg, ARMword *data)
```

where:

mdesc	is the handle for the core.
cpnum	is the number of the coprocessor.
reg	is the number of the coprocessor register to read from, as indexed by `CR`*n* in an `LDC` or `STC` instruction.
data	is a pointer for the data read from the coprocessor register. The number of words transferred, and the order of the words, is coprocessor dependent.

Return

The function must return:

- `ARMul_DONE`, if the register can be written
- `ARMul_CANT`, if the register cannot be written.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.3 Basic model interface

This section has the following subsections:

- *Declaration of a private state data structure*
- *Model initialization*
- *Model finalization*.

For each model, you must write an initialization function. For additional functionality, you must register callbacks.

Macros are provided in `minperip.h` for the following abstractions:

- *Declaration of a private state data structure* on page 4-11
- *Model initialization* on page 4-12
- *Model finalization* on page 4-12.

See also *Initialization, finalization, and state macros* on page 3-6.

4.3.1 Declaration of a private state data structure

Each model must store its state in a private data structure. Initialization and finalization macros are provided by `ARMulif`. These macros require the use of certain fields in this data structure.

To declare a state data structure, use the `BEGIN_STATE` and `END_STATE` macros as follows:

```
/*
 * Create a YourModelState data structure
 */
BEGIN_STATE_DECL(YourModel)
/*
 * Your private data here
 */
END_STATE_DECL(YourModel)
```

This declares a structure:

```
typedef struct YourModelState
```

This structure contains:

- predefined data fields:
  - `toolconf config`
  - `const struct RDI_HostosInterface *hostif`
  - `RDI_ModuleDesc coredesc;`
  - `RDI_ModuleDesc agentdesc`
- the private data you put between the macros.

4.3.2 Model initialization

The `BEGIN_INIT()` and `END_INIT()` macros delineate the initialization function for the model. The initialization function is called:

- during ARMulator initialization
- whenever a new image is downloaded from the debugger.

The following local variables are provided in the initialization function:

- **`bool`** `coldboot`

  `TRUE` if ARMulator is initializing, `FALSE` if a new image is being downloaded from the debugger.

- `YourModelState *state`

  A pointer to the private state data structure. Memory for this is allocated and cleared by the initialization macro, and the predefined data fields are initialized.

In the initialization function, your model must:

- initialize any private data
- install any callbacks.

4.3.3 Model finalization

The `BEGIN_EXIT()` and `END_EXIT()` macros delineate the finalization function for the model. The finalization function is called when ARMulator is closing down.

The following local variable is provided in the finalization function:

`YourModelState *state`

Your model must de-install any callbacks in the finalization function.

The `END_EXIT()` macro frees memory allocated for state.

4.4 Coprocessor model interface

The coprocessor model interface is defined in `armul_copro.h`. The basic coprocessor functions are:

- *ARMulif_InstallCoprocessorV5* on page 4-14
- *LDC* on page 4-15
- *STC* on page 4-16
- *MRC* on page 4-17
- *MCR* on page 4-18
- *MRC* on page 4-17
- *MCR* on page 4-18
- *MCRR* on page 4-19
- *MRRC* on page 4-20
- *CDP* on page 4-21.

In addition, two functions are provided that enable a debugger to read and write coprocessor registers through the *Remote Debug Interface* (RDI). They are:

- *read* on page 4-22
- *write* on page 4-23.

If a coprocessor does not handle one or more of these functions, it must leave their entries in the `ARMul_CPInterface` structure unchanged.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.4.1 ARMulif_InstallCoprocessorV5

Use this function to register a coprocessor handler.

Syntax

```
unsigned ARMulif_InstallCoprocessorV5(RDI_ModuleDesc *mdesc, unsigned number,
                                      struct ARMiss_CoprocessorV5 *cpv5, void *handle)
```

where:

mdesc	is the handle for the core.
number	is the coprocessor number.
cpv5	is a pointer to the coprocessor interface structure.
handle	is a pointer to private data to pass to each coprocessor function.

Return

This function returns either:

- `ARMulErr_NoError`, if there is no error
- an `ARMul_Error` value.

See `armerrs.h` and `errors.h` for a full list of error codes. The error must be passed through `Hostif_RaiseError()` for formatting (see *Hostif_RaiseError* on page 4-39).

4.4.2 LDC

This function is called when an `LDC` instruction is recognized for a coprocessor.

Syntax

`unsigned` `LDC(`**`void`** `*`*`handle`*`,` **`unsigned`** *`type`*`, ARMword` *`instr`*`, ARMword` *`data`*`)`

where:

`handle`	is the handle from `ARMulif_InstallCoprocessorV5`.
`type`	is the type of coprocessor access. This can be one of:

`ARMul_FIRST`	indicates that this is the first time the coprocessor model has been called for this instruction.
`ARMul_BUSY`	indicates that this is a subsequent call, after the first call was busy-waited.
`ARMul_INTERRUPT`	warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction will be retried later, in which case the *`type`* will be reset to `ARMul_FIRST`.
`ARMul_TRANSFER`	indicates that the ARM is about to perform the load.
`ARMul_DATA`	indicates that valid data is included in *`data`*.

`instr`	the current opcode.
`data`	is the data being transferred to the coprocessor.

Return

The function must return one of:

- `ARMul_INC`, to request more data from the core (only in response to `ARMul_FIRST`, `ARMul_BUSY`, or `ARMul_DATA`)
- `ARMul_DONE`, to indicate that the coprocessor operation is complete (only in response to `ARMul_DATA`)
- `ARMul_BUSY`, to indicate that the coprocessor is busy (only in response to `ARMul_FIRST` or `ARMul_BUSY`)
- `ARMul_CANT`, to indicate that the instruction is not supported, or the specified register cannot be accessed (only in response to `ARMul_FIRST` or `ARMul_BUSY`).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.4.3 STC

This function is called when an `STC` instruction is recognized for a coprocessor.

Syntax

`unsigned` `STC(`**`void`** `*`*`handle`*`, `**`unsigned`** *`type`*`, ARMword `*`instr`*`, ARMword *`*`data`*`)`

where:

handle	is the handle from `ARMulif_InstallCoprocessorV5`.
type	is the type of the coprocessor access. This can be one of:

`ARMul_FIRST`	indicates that this is the first time the coprocessor model has been called for this instruction.
`ARMul_BUSY`	indicates that this is a subsequent call, after the first call was busy-waited.
`ARMul_INTERRUPT`	warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction will be retried later, in which case the *`type`* will be reset to `ARMul_FIRST`.
`ARMul_DATA`	indicates that the coprocessor must return valid data in `*`*`data`*.

instr	is the current opcode.
data	is a pointer to the location of the data being transferred from the coprocessor to the core.

Return

The function must return one of:

- `ARMul_INC`, to indicate that there is more data to transfer to the core (only in response to `ARMul_FIRST`, `ARMul_BUSY`, or `ARMul_DATA`)
- `ARMul_DONE`, to indicate that the coprocessor operation is complete (only in response to `ARMul_DATA`)
- `ARMul_BUSY`, to indicate that the coprocessor is busy (only in response to `ARMul_FIRST` or `ARMul_BUSY`)
- `ARMul_CANT`, to indicate that the instruction is not supported, or the specified register cannot be accessed (only in response to `ARMul_FIRST` or `ARMul_BUSY`).

4.4.4 MRC

This function is called when an `MRC` instruction is recognized for a coprocessor. If the requested coprocessor register does not exist or cannot be written to, the function must return `ARMul_CANT`.

Syntax

`unsigned` `MRC(`**`void`** `*`*`handle`*`,` **`unsigned`** *`type`*`, ARMword` *`instr`*`, ARMword *`*`data`*`)`

where:

`handle`	is the handle from `ARMulif_InstallCoprocessorV5`.
`type`	is the type of the coprocessor access. This can be one of:
	`ARMul_FIRST` indicates that this is the first time the coprocessor model has been called for this instruction.
	`ARMul_BUSY` indicates that this is a subsequent call, after the first call was busy-waited.
	`ARMul_INTERRUPT` warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction will be retried later, in which case the *`type`* will be reset to `ARMul_FIRST`.
	`ARMul_DATA` indicates that valid data is included in `*`*`data`*.
`instr`	is the current opcode.
`data`	is a pointer to the location of the data being transferred from the coprocessor to the core.

Return

The function must return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete, and valid data has been returned to `*`*`data`*.
- `ARMul_BUSY`, to indicate that the coprocessor is busy
- `ARMul_CANT`, to indicate that the instruction is not supported, or the specified register cannot be accessed.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.4.5 MCR

This function is called when an `MCR` instruction is recognized for a coprocessor. If the requested coprocessor register does not exist or cannot be written to, the function must return `ARMul_CANT`.

Syntax

unsigned MCR(**void** **handle*, **unsigned** *type*, ARMword *instr*, ARMword *data*)

where:

handle	is the handle from `ARMulif_InstallCoprocessorV5`.
type	is the type of the coprocessor access. This can be one of:

`ARMul_FIRST`	indicates that this is the first time the coprocessor model has been called for this instruction.
`ARMul_BUSY`	indicates that this is a subsequent call, after the first call was busy-waited.
`ARMul_INTERRUPT`	warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction will be retried later, in which case the *type* will be reset to `ARMul_FIRST`.
`ARMul_DATA`	indicates valid data is included in *data*.

instr	is the current opcode.
data	is the data being transferred to the coprocessor.

Return

The function must return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete
- `ARMul_BUSY`, to indicate that the coprocessor is busy
- `ARMul_CANT`, to indicate that the instruction is not supported, or the specified register cannot be accessed.

4.4.6 MCRR

This function is called when an `MCRR` instruction is recognized for a coprocessor.

The function must return `ARMul_CANT` if:

- the requested coprocessor register does not exist
- the requested coprocessor register cannot be written to
- the coprocessor is ARM architecture v4T or earlier.

Syntax

```
unsigned MCRR(void *handle, unsigned type, ARMword instr, ARMword data)
```

where:

handle is the handle from `ARMulif_InstallCoprocessorV5`.

type is the type of the coprocessor access. This can be one of:

`ARMul_FIRST`	indicates that this is the first time the coprocessor model has been called for this instruction.
`ARMul_BUSY`	indicates that this is a subsequent call, after the first call was busy-waited.
`ARMul_INTERRUPT`	warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction will be retried later, in which case the *type* will be reset to `ARMul_FIRST`.
`ARMul_DATA`	indicates valid data is included in *data*.

instr is the current opcode.

data is the data being transferred to the coprocessor.

Return

The function must return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete
- `ARMul_BUSY`, to indicate that the coprocessor is busy
- `ARMul_CANT`, to indicate that the instruction is not supported, or the specified register cannot be accessed.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.4.7 MRRC

This function is called when an `MCR` instruction is recognized for a coprocessor.

The function must return `ARMul_CANT` if:

- the requested coprocessor register does not exist
- the requested coprocessor register cannot be read from
- the coprocessor is ARM architecture v4T or earlier.

Syntax

`unsigned` `MRRC(`**`void`** `*`*`handle`*`,` **`unsigned`** *`type`*`, ARMword` *`instr`*`, ARMword` *`data`*`)`

where:

`handle`	is the handle from `ARMulif_InstallCoprocessorV5`.
`type`	is the type of the coprocessor access. This can be one of:

`ARMul_FIRST`	indicates that this is the first time the coprocessor model has been called for this instruction.
`ARMul_BUSY`	indicates that this is a subsequent call, after the first call was busy-waited.
`ARMul_INTERRUPT`	warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction will be retried later, in which case the *`type`* will be reset to `ARMul_FIRST.`
`ARMul_DATA`	indicates valid data is included in *`data`*.

`instr`	is the current opcode.
`data`	is the data being transferred from the coprocessor.

Return

The function must return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete
- `ARMul_BUSY`, to indicate that the coprocessor is busy
- `ARMul_CANT`, to indicate that the instruction is not supported, or the specified register cannot be accessed.

4.4.8 CDP

This function is called when a `CDP` instruction is recognized for a coprocessor. If the requested coprocessor operation is not supported, the function must return `ARMul_CANT`.

Syntax

```
unsigned CDP(void *handle, unsigned type, ARMword instr, ARMword *data)
```

where:

handle	is the handle from `ARMulif_InstallCoprocessorV5`.
type	is the type of the coprocessor access. This can be one of:
	`ARMul_FIRST` indicates that this is the first time the coprocessor model has been called for this instruction.
	`ARMul_BUSY` indicates that this is a subsequent call, after the first call was busy-waited.
	`ARMul_INTERRUPT` warns the coprocessor that the ARM is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction will be retried later, in which case the *type* will be reset to `ARMul_FIRST`.
instr	is the current opcode.
data	is not used.

Return

The function must return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete
- `ARMul_BUSY`, to indicate that the coprocessor is busy
- `ARMul_CANT`, to indicate that the instruction is not supported.

4.4.9 read

This function enables a debugger to read a coprocessor register. The function reads the coprocessor register numbered *reg* and transfers its value to the location addressed by *value*.

If the requested coprocessor register does not exist, or the register cannot be read, the function must return ARMul_CANT.

Syntax

```
unsigned read(void *handle, unsigned reg, ARMword instr, ARMword *value)
```

where:

handle	is the handle from ARMulif_InstallCoprocessorV5.
reg	is the register number of the coprocessor register to be read.
instr	is not used.
value	is a pointer to the location of the data to be read from the coprocessor by RDI.

Return

The function must return one of:

- ARMul_DONE, to indicate that the coprocessor operation is complete
- ARMul_CANT, to indicate that the register is not supported.

Usage

This function is called by the debugger.

4.4.10 write

This function enables a debugger to write to a coprocessor register.

Syntax

`unsigned` `write(`**`void`** `*`*`handle`*`,` **`unsigned`** *`reg`*`, ARMword` *`instr`*`, ARMword *`*`value`*`)`

where:

`handle`	is the handle from `ARMulif_InstallCoprocessorV5`.
`reg`	is the register number of the coprocessor register that is to be written.
`instr`	is not used.
`value`	is a pointer to the location of the data that is to be written to the coprocessor.

Return

The function must return one of:

- `ARMul_DONE`, to indicate that the coprocessor operation is complete
- `ARMul_CANT`, to indicate that the register is not supported.

Usage

This function is called by the debugger.

The function writes the value at the location addressed by *`value`* to the coprocessor register numbered *`reg`*.

If the requested coprocessor does not exist or the register cannot be written, the function must return `ARMul_CANT`.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.5 Exceptions

The following functions enable a model to set or clear signals:

- *ARMulif_SetSignal* on page 4-24
- *ARMulif_GetProperty* on page 4-25.

4.5.1 ARMulif_SetSignal

The `ARMulif_SetSignal` function is used to set the state of signals or properties.

Syntax

```
void ARMulif_SetSignal(RDI_ModuleDesc *mdesc, ARMSignalType sigType,
                       SignalState sigState)
```

where:

mdesc is the handle for the core.

sigtype is the signal to be set. *sigtype* can be any one of:

`RDIPropID_ARMSignal_IRQ`
Assert an interrupt.

`RDIPropID_ARMSignal_FIQ`
Assert a fast interrupt.

`RDIPropID_ARMSignal_Reset`
Assert the reset signal. The core will reset, and will not restart until the reset signal is de-asserted.

`RDIPropID_ARMSignal_BigEnd`
Set this signal for big-endian operation, or clear it for little-endian operation.

`RDIPropID_ARMSignal_HighException`
Set the base location of exception vectors.

`RDIPropID_ARMSignal_BranchPredictEnable`
(ARM10 only)

`RDIPropID_ARMSignal_LDRSetTBITDisable`
(ARM10 only)

`RDIPropID_ARMSignal_WaitForInterrupt`
(ARM10 and XScale only)

`RDIPropID_ARMSignal_DebugState`
Enter or exit debug state.

	`RDIPropID_ARMulProp_CycleDelta`	
		Wait the core for a specified number of cycles.
	`RDIPropID_ARMulProp_Accuracy`	
		Select the modelling accuracy, as a percentage in the range 0% to 100%.
`sigstate`	For signals, you must give *`sigstate`* one of the following values:	
	`FALSE`	Signal off
	`TRUE`	Signal on.
	For properties, you must give *`sigstate`* an integer value.	

Note

For information about signalling interrupts when using an interrupt controller see *Interrupt controller* on page 4-67.

4.5.2 ARMulif_GetProperty

The `ARMulif_GetProperty` function is used to read the values of properties and signals.

Syntax

```
void ARMulif_GetProperty(RDI_ModuleDesc *mdesc, ARMSignalType id,
                         ARMword *value)
```

where:

`mdesc`	is the handle for the core.	
`id`	is the signal or property to read. *`id`* can be any one of:	
	`RDIPropID_ARMSignal_IRQ`	
		`TRUE` if the interrupt signal is asserted.
	`RDIPropID_ARMSignal_FIQ`	
		`TRUE` if the fast interrupt signal is asserted.
	`RDIPropID_ARMSignal_Reset`	
		`TRUE` if the reset signal is asserted.
	`RDIPropID_ARMSignal_BigEnd`	
		`TRUE` if the bigend signal is asserted.
	`RDIPropID_ARMSignal_HighException`	
		`TRUE` if the fast interrupt signal is asserted.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

`RDIPropID_ARMSignal_BranchPredictEnable`
: (ARM10 only)

`RDIPropID_ARMSignal_LDRSetTBITDisable`
: (ARM10 only)

`RDIPropID_ARMSignal_WaitForInterrupt`
: (ARM10 and XScale only)

`RDIPropID_ARMulProp_CycleCount`
: Count of the number of cycles executed since initialization.

`RDIPropID_ARMulProp_RDILog`
: Current setting of the RDI log level. Generally, this is zero if logging is disabled, and nonzero if it is enabled.

`RDIPropID_ARMSignal_ProcessorProperties`
: The properties word associated with the processor being simulated. This is a bitfield of properties, defined in `armdefs.h`.

`value` is a pointer to a block to write the property to. This allows for properties with more than 32 bits. However, all the properties listed are actually 32 bits wide at most.

4.6 Events

ARMulator has a mechanism for broadcasting and handling events. These events consist of an event number and a pair of words. The number identifies the event. The semantics of the words depends on the event.

The core ARMulator generates some example events, defined in `armdefs.h`. They are divided into three groups:

- events from the ARM processor core, listed in Table 4-3 on page 4-28
- events from the MMU and cache (not on StrongARM®-110), listed in Table 4-2 on page 4-27
- events from the prefetch unit (ARM8™-based processors only), listed in Table 4-4 on page 4-28
- configuration change events, listed in Table 4-6 on page 4-29.

These events can be logged in the trace file if tracing is enabled, and trace events is turned on. Additional modules can provide new event types that will be handled in the same way. User defined events must have values between `UserEvent_Base` (`0x100000`) and `UserEvent_Top` (`0x1FFFFF`).

You can catch events by installing an event handler (see *Event handler* on page 4-35). You can raise an event by calling `ARMulif_RaiseEvent()` (see *ARMulif_RaiseEvent* on page 4-30).

Table 4-2 Events from the MMU and cache (not on StrongARM-110)

Event name	Word 1	Word 2	Event number
`MMUEvent_DLineFetch`	Miss address	Victim address	`0x10001`
`MMUEvent_ILineFetch`	Miss address	Victim address	`0x10002`
`MMUEvent_WBStall`	Physical address of write	Number of words in write buffer	`0x10003`
`MMUEvent_DTLBWalk`	Miss address	Victim address	`0x10004`
`MMUEvent_ITLBWalk`	Miss address	Victim address	`0x10005`
`MMUEvent_LineWB`	Miss address	Victim address	`0x10006`
`MMUEvent_DCacheStall`	Address causing stall	Address fetching	`0x10007`
`MMUEvent_ICacheStall`	Address causing stall	Address fetching	`0x10008`

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Table 4-3 Events from the ARM processor core

Event name	Word 1	Word 2	Event number
`CoreEvent_Reset`	-	-	`0x1`
`CoreEvent_UndefinedInstr`	pc value	Instruction	`0x2`
`CoreEvent_SWI`	pc value	SWI number	`0x3`
`CoreEvent_PrefetchAbort`	pc value	-	`0x4`
`CoreEvent_DataAbort`	pc value	Aborting address	`0x5`
`CoreEvent_AddrExceptn`	pc value	Aborting address	`0x6`
`CoreEvent_IRQ`	pc value	-	`0x7`
`CoreEvent_FIQ`	pc value	-	`0x8`
`CoreEvent_Breakpoint`	pc value	RDI_PointHandle	`0x9`
`CoreEvent_Watchpoint`	pc value	Watch address	`0xA`
`CoreEvent_IRQSpotted`	pc value	-	`0x17`
`CoreEvent_FIQSpotted`	pc value	-	`0x18`
`CoreEvent_ModeChange`	pc value	New mode	`0x19`
`CoreEvent_Dependency`	pc value	Interlock register bitmask	`0x20`

Table 4-4 Events from the prefetch unit (ARM810 only)

Event name	Word 1	Word 2	Event number
`PUEvent_Full`	Next pc value	-	`0x20001`
`PUEvent_Mispredict`	Address of branch	-	`0x20002`
`PUEvent_Empty`	Next pc value	-	`0x20003`

Table 4-5 Debug events

Event name	Word 1	Word 2	Event number
`DebugEvent_InToDebug`	-	-	`0x40001`
`DebugEvent_OutOfDebug`	-	-	`0x40002`
`DebugEvent_DebuggerChangedPC`	pc	-	`0x40003`

Table 4-6 Config events

Event name	Word 1	Word 2	Event number
`ConfigEvent_AllLoaded`	-	-	`0x50001`
`ConfigEvent_Reset`	-	-	`0x50002`
`ConfigEvent_VectorsLoaded`	-	-	`0x50003`
`ConfigEvent_EndiannessChanged`	1 (big end) or 2 (little end)	-	`0x50005`

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.6.1 ARMulif_RaiseEvent

This function invokes events. The events are passed to the user-supplied event handlers.

Syntax

```
void ARMulif_RaiseEvent(RDI_ModuleDesc *mdesc, ARMword event,
                        ARMword data1, ARMword data2)
```

where:

mdesc	is the handle for the core.
event	is one of the event numbers defined in Table 4-2 on page 4-27, Table 4-3 on page 4-28, Table 4-4 on page 4-28, or Table 4-5 on page 4-29.
data1	is the first word of the event.
data2	is the second word of the event.

4.7 Handlers

ARMulator can be made to call back your model when some state values change. You do this by installing the relevant *event handler*.

You must provide implementations of the event handlers if you want to use them in your own models. See the implementations in the ARM supplied models for examples.

You can use event handlers to avoid having to check state values on every access. For example, a peripheral model is expected to present the ARM core with data in the correct byte order for the value of the ARM processor **bigend** signal. A peripheral model can attach to the `EventHandler()` (see *Event handler* on page 4-35) to be informed when this signal changes.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.7.1 Exception handler

This event handler is called whenever the ARM processor takes an exception.

Syntax

```
typedef unsigned GenericCallbackFunc(void *handle, void *data)
```

where:

handle is the handle passed to `ARMulif_InstallExceptionHandler`.

data must be cast to (`ARMul_Event *`), and contain:

`((ARMul_Event *)data)->event`
: is the core event causing the exception (see Table 4-3 on page 4-28).

`((ARMul_Event *)data)->data1`
: is the address of the hardware vector for the exception.

`((ARMul_Event *)data)->data2`
: is the instruction that caused the exception.

Usage

As an example, this can be used by an operating system model to intercept and simulate SWIs. If an installed handler returns nonzero, the ARM does not take the exception (the exception is ignored).

Note

If the processor is in Thumb state, the equivalent ARM instruction will be supplied.

Install the exception handler using:

```
int ARMulif_InstallExceptionHandler(RDI_ModuleDesc *mdesc,
                                    GenericCallbackFunc *func, void *handle)
```

Remove the exception handler using:

```
int ARMulif_RemoveExceptionHandler(RDI_ModuleDesc *mdesc,
                                   GenericCallbackFunc *func, void *handle)
```

4.7.2 Unknown RDI information handler

The unknown RDI information function is called if ARMulator cannot handle an `RDI_InfoProc` request itself. It returns an `RDIError` value. This function can be used by a model extending the RDI interface between ARMulator and the debugger. For example, the profiler module (in `profiler.c`) provides the `RDIProfile` info calls.

Syntax

```
typedef int RDI_InfoProc(void *handle, unsigned type,
                         ARMword *arg1, ARMword *arg2)
```

where:

handle	is the handle passed to `ARMulif_InstallUnkRDIInfoHandler`.
type	is the `RDI_InfoProc` subcode. These are defined in `rdi_info.h`. See below for some examples.
arg1/arg2	are arguments passed to the handler from ARMulator.

Usage

ARMulator stops calling `RDI_InfoProc()` functions when one returns a value other than `RDIError_UnimplementedMessage`.

The following codes are examples of the `RDI_InfoProc` subcodes that can be specified as *type*:

`RDIInfo_Target`

This enables models to declare how to extend the functionality of the target. For example, `profiler.c` intercepts this call to set the `RDITarget_CanProfile` flag.

`RDIInfo_SetLog`

This is passed around so that models can switch logging information on and off. For example, `tracer.c` uses this call to switch tracing on and off from bit 4 of the `rdi_log` value.

`RDIRequestCyclesDesc`

This enables models to extend the list of counters provided by the debugger in `$statistics`. Models call `ARMul_AddCounterDesc()` (see *General purpose functions* on page 4-39) to declare each counter in turn. It is essential that the model also trap the `RDICycles` RDI info call.

`RDICycles`	Models that have declared a statistics counter by trapping `RDIRequestCyclesDesc` must also respond to `RDICycles` by calling `ARMul_AddCounterValue()` (see *General purpose functions* on page 4-39) for each counter in turn, in the same order as they were declared.

The above RDI info calls have already been dealt with by ARMulator, and are passed for information only, or so that models can add information to the reply. Models must always respond to these messages with `RDIError_UnimplementedMessage`, so that the message is passed on even if the model has responded.

Install the handler using:

```
int ARMulif_InstallUnkRDIInfoHandler(RDI_ModuleDesc *mdesc,
                                     RDI_InfoProc *func, void *handle)
```

Remove the handler using:

```
int ARMulif_RemoveUnkRDIInfoHandler(RDI_ModuleDesc *mdesc,
                                    RDI_InfoProc *func, void *handle)
```

Example

The `semihost.c` model supplied with ARMulator uses the `UnkRDIInfoUpcall()` to interact with the debugger:

`RDIErrorP`	returns errors raised by the program running under ARMulator to the debugger.
`RDISet_Cmdline`	finds the command line set for the program by the debugger.
`RDIVector_Catch`	intercepts the hardware vectors.

4.7.3 Event handler

This handler catches ARMulator events (see *Events* on page 4-27).

Syntax

```
typedef unsigned GenericCallbackFunc(void *handle, void *data)
```

where:

handle is the handle passed to `ARMulif_InstallEventHandler`.

data must be cast to (`ARMul_Event *`), and contain:

`((ARMul_Event *)data)->event`
is one of the event numbers defined in Table 4-2 on page 4-27, Table 4-3 on page 4-28, and Table 4-4 on page 4-28.

`((ARMul_Event *)data)->addr1`
is the first word of the event.

`((ARMul_Event *)data)->addr2`
is the second word of the event.

Usage

Install the handler using:

```
void *ARMulif_InstallEventHandler(RDI_ModuleDesc *mdesc, uint32 events,
                                  GenericCallbackFunc *func, void *handle)
```

Specify one or more of the following for *events*:

- `CoreEventSel`
- `MMUEventSel`
- `PUEventSel`
- `DebugEventSel`
- `TraceEventSel`
- `ConfigEventSel`.

Remove the handler using:

```
int ARMulif_RemoveEventHandler(RDI_ModuleDesc *mdesc, void *node)
```

Example handler installation

```
ARMulif_InstallEventHandler(mdesc, CoreEventSel | ConfigEventSel, func, handle)
```

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.8 Memory access functions

The memory system can be probed by a peripheral model using a set of functions for reading and writing memory. These functions access memory without inserting cycles on the bus. If your model inserts cycles on the bus, it must install itself as a memory model, possibly between the core and the real memory model.

Note

It is not possible to tell if these calls result in a data abort.

4.8.1 Reading from a given address

The following functions return the word, halfword, or byte at the specified address. Each function accesses the memory without inserting cycles on the bus.

Syntax

```
ARMword ARMulif_ReadWord(RDIModuleDesc *mdesc, ARMword address)
ARMword ARMulif_ReadByte(RDIModuleDesc *mdesc, ARMword address)
```

where:

mdesc	is the handle for the core.
address	is the address in simulated memory from which the word, halfword, or byte is to be read.

Return

The functions return the word or byte, as appropriate.

4.8.2 Writing to a specified address

The following functions write the specified word, halfword, or byte at the specified address. Each function accesses memory without inserting cycles on the bus.

Syntax

```
void ARMulif_WriteWord(RDIModuleDesc *mdesc, ARMword address, ARMword data)
```

```
void ARMulif_WriteByte(RDIModuleDesc *mdesc, ARMword address, ARMword data)
```

where:

mdesc	is the handle for the core.
address	is the address in simulated memory to write to.
data	is the word or byte to write.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.9 Event scheduling functions

The following functions enable you to schedule or remove events:

- *ARMulif_ScheduleTimedFunction* on page 4-38
- *ARMulif_DescheduleTimedFunction* on page 4-38.

4.9.1 ARMulif_ScheduleTimedFunction

This function schedules events using memory system cycles. It enables a function to be called at a specified number of cycles in the future.

Syntax

```
void *ARMulif_ScheduleTimedFunction(RDI_ModuleDesc *mdesc,
                                    ARMul_TimedCallback *tcb)
```

where:

mdesc	is the handle for the core.
tcb	is a handle for you to use if you want to deschedule the function.

Note

The function can be called only on the first instruction boundary following the specified cycle.

4.9.2 ARMulif_DescheduleTimedFunction

`ARMul_DescheduleTimedFunction()` removes a previously-scheduled memory cycle based event.

Syntax

```
unsigned ARMulif_DescheduleTimedFunction(RDI_ModuleDesc *mdesc, void *tcb);
```

where:

mdesc	is the handle for the core.
tcb	is the handle supplied by `ARMulif_ScheduleTimedFunction` when the even was first set up.

4.10 General purpose functions

The following are general purpose ARMulator functions. They include functions to access processor properties, add counter descriptions and values, stop ARMulator and execute code:

- *Hostif_RaiseError*
- *ARMulif_Time* on page 4-40
- *ARMulif_IncurTime* on page 4-40
- *ARMulif_CondCheckInstr* on page 4-41
- *ARMul_AddCounterDesc* on page 4-42
- *ARMul_AddCounterValue* on page 4-43
- *ARMulif_StopExecution* on page 4-45
- *ARMulif_EndCondition* on page 4-45.

4.10.1 Hostif_RaiseError

Several initialization and installation functions can return errors of type `ARMul_Error`. These errors must be passed through `Hostif_RaiseError()`. This is a printf-like function that formats the error message associated with an `ARMul_Error` error code.

`Hostif_RaiseError` only prints the error message. After calling this function, the model must return with an appropriate error, such as `RDIError_UnableToInitialise`.

`Hostif_RaiseError` must only be used during initialization.

Syntax

```
void Hostif_RaiseError(const struct RDI_HostosInterface *hostif,
                       const char *format, ...)
```

where:

hostif	is the handle for the host interface.
format	is the error code for the error message to be formatted.
`...`	are printf-style format specifiers of variadic type.

4.10.2 ARMulif_Time

This function returns the number of memory cycles executed since system reset.

Syntax

```
ARMTime ARMulif_Time(RDIModuleDesc *mdesc)
```

where:

mdesc	is the handle for the core.

Return

The function returns the total number of cycles executed since system reset.

4.10.3 ARMulif_IncurTime

This function increments the cycle counter by a specified number of clock cycles.

Syntax

```
void ARMulif_IncurTime(RDI_ModuleDesc *mdesc, ARMTime number)
```

where:

mdesc	is the handle for the core.
number	is the number of cycles to increment the counter by.

4.10.4 ARMulif_CondCheckInstr

This function tests an instruction opcode against the current state of the PSR flags.

Syntax

```
unsigned ARMulif_CondCheckInstr(RDI_ModuleDesc *mdesc, ARMword instr)
```

where:

mdesc	is the handle for the core.
instr	is the instruction opcode to check.

Return

This function returns:

- 1, if the instruction would execute
- 0, if the instruction would not execute.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.10.5 ARMul_AddCounterDesc

The `ARMul_AddCounterDesc()` function adds new counters to `$statistics`.

Syntax

```
int ARMul_AddCounterDesc(void *handle, ARMword *arg1, ARMword *arg2,
                         const char *name)
```

where:

handle	is no longer used.
arg1/*arg2*	are the arguments passed to the `UnkRDIInfoUpcall()`.
name	is a string that names the statistic counter. The string must be less than 32 characters long.

Return

The function returns one of:

- `RDIError_BufferFull`
- `RDIError_UnimplementedMessage`.

Usage

When ARMulator receives an `RDIRequestCycleDesc()` call from the debugger, it uses the `UnkRDIInfoUpcall()` (see *Unknown RDI information handler* on page 4-33) to ask each module in turn if it wishes to provide any statistics counters. Each module responds by calling `ARMul_AddCounterDesc()` with the arguments passed to the `UnkRDIInfoUpcall()`.

All statistics counters must be either a 32-bit or 64-bit word, and be monotonically increasing. That is, the statistic value must go up over time. This is a requirement because of the way the debugger calculates `$statistics_inc`.

4.10.6 ARMul_AddCounterValue

This function provides the facility for your model to supply statistics for the debugger to display.

Syntax

```
int ARMul_AddCounterValue(void *handle, ARMword *arg1, ARMword *arg2, bool is64,
                          const ARMword *counter)
```

where:

handle	is no longer used.
arg1/*arg2*	are the arguments passed to the `UnkRDIInfoUpcall()`.
is64	denotes whether the counter is a pair of 32-bit words making a 64-bit counter (least significant word first), or a single 32-bit value. This enables modules to provide a full 64-bit counter.
counter	is a pointer to the current value of the counter.

Return

The function always returns `RDIError_UnimplementedMessage`.

Usage

Your model must call this function, or `ARMul_AddCounterValue64`, from its `UnkRDIInfoUpcall()` handler. `ARMul_AddCounterValue64` is identical to `ARMul_AddCounterValue` except for the word order of the counter.

4.10.7 ARMul_AddCounterValue64

This function provides the facility for your model to supply statistics for the debugger to display.

Syntax

```
int ARMul_AddCounterValue64(void *handle, ARMword *arg1, ARMword *arg2,
                            const uint64 counterval)
```

where:

handle	is no longer used.
arg1/*arg2*	are the arguments passed to the `UnkRDIInfoUpcall()`.
counterval	is the current value of the counter.

Return

The function always returns `RDIError_UnimplementedMessage`.

Usage

Your model must call this function, or `ARMul_AddCounterValue`, from its `UnkRDIInfoUpcall()` handler. This function is identical to `ARMul_AddCounterValue` except that the word order is big-endian or little-endian according to the word order of the host system.

4.10.8 ARMulif_StopExecution

This function stops simulator execution at the end of the current instruction, giving a reason code.

Syntax

void ARMulif_StopExecution(RDI_ModuleDesc **mdesc*, **unsigned** *reason*)

where:

mdesc	is the handle for the core.
reason	is an RDIError error value. The debugger interprets *reason* and issues a suitable message. Expected errors are:

RDIError_NoError	Program ran to a natural termination.
RDIError_BreakpointReached	Stop condition was a breakpoint.
RDIError_WatchPointReached	Stop condition was a watchpoint.
RDIError_UserInterrupt	Execution interrupted by the user.

4.10.9 ARMulif_EndCondition

This function returns the *reason* passed to ARMulif_StopExecution.

Syntax

unsigned ARMulif_EndCondition(RDI_ModuleDesc **mdesc*)

where:

mdesc	is the handle for the core.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.11 Accessing the debugger

This section describes the input, output, and RDI functions that you can use to access the debugger.

Several functions are provided to display messages in the host debugger. Under `armsd`, these functions print messages to the console. Under AXD, ADW, or ADU they display messages to the relevant window:

- *Hostif_DebugPrint*
- *Hostif_ConsolePrint* on page 4-47
- *Hostif_PrettyPrint* on page 4-47
- *Hostif_DebugPause* on page 4-50.

All of these functions take the following as the first parameter:

```
const struct RDI_HostosInterface *hostif
```

This value is available in the state datastructure of the model, as defined between the `BEGIN_STATE_DECL()` and `END_STATE_DECL()` macros (see *Basic model interface* on page 4-11).

4.11.1 Hostif_DebugPrint

This function displays a message in the RDI logging window under a GUI debugger, or to the console under `armsd`.

Syntax

```
void Hostif_DebugPrint(const struct RDI_HostosInterface *hostif,
                       const char *format, ...)
```

where:

`hostif`	is the handle for the host interface.
`format`	is a pointer to a printf-style formatted output string.
`...`	are a variable number of parameters associated with *`format`*.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.11.2 Hostif_ConsolePrint

This function prints the text specified in the format string to the ARMulator console. Under AXD, ADW, or ADU, the text appears in the console window.

Syntax

```
void Hostif_ConsolePrint(const struct RDI_HostosInterface *hostif,
                         const char *format, ...)
```

where:

hostif	is the handle for the host interface.
format	is a pointer to a printf-style formatted output string.
`...`	are a variable number of parameters associated with *format*.

Note

Use `Hostif_PrettyPrint()` to display startup messages.

4.11.3 Hostif_PrettyPrint

This function prints a string in the same way as `Hostif_ConsolePrint()`, but in addition performs line-break checks so that wordwrap is avoided. Use it to display startup messages.

Syntax

```
void Hostif_PrettyPrint(const struct RDI_HostosInterface *hostif,
                        struct hashblk * /*toolconf*/ config,
                        const char *format, ...)
```

where:

hostif	is the handle for the host interface.
config	is a pointer to the toolconf configuration database of the model. This value is available in the state datastructure of the model, as defined between the `BEGIN_STATE_DECL()` and `END_STATE_DECL()` macros (see *Basic model interface* on page 4-11).
format	is a pointer to a printf-style formatted output string.
`...`	are a variable number of parameters associated with *format*.

4.11.4 Hostif_ConsoleReadC

This function reads a character from the ARMulator console.

Syntax

```
int Hostif_ConsoleReadC(const struct
                        RDI_HostosInterface *hostif)
```

where:

hostif	is the handle for the host interface.

Return

This function returns the ASCII value of the character read, or EOF.

4.11.5 Hostif_WriteC

This function writes a character to the ARMulator console.

Syntax

```
void Hostif_ConsoleWriteC(const struct
                          RDI_HostosInterface *hostif, int c)
```

where:

hostif	is the handle for the host interface.
c	is the character to write. *c* is converted to an unsigned char.

4.11.6 Hostif_ConsoleRead

This function reads a string from the ARMulator console. Reading terminates at a newline or if the end of the buffer is reached.

Syntax

```
char *Hostif_ConsoleRead(const struct RDI_HostosInterface *hostif,
                         char *buffer, int len)
```

where:

hostif	is the handle for the host interface.
buffer	is a pointer to a buffer to hold the string.
len	is the maximum length of the buffer.

Return

This function returns a pointer to a buffer, or NULL on error or end of file.

The buffer contains at most `len-1` characters, terminated by a zero. If a newline is read, it is included in the string before the zero.

4.11.7 Hostif_ConsoleWrite

This function writes a string to the ARMulator console.

Syntax

```
int Hostif_ConsoleWrite(const struct RDI_HostosInterface *hostif,
                        const char *buffer, int len)
```

where:

hostif	is the handle for the host interface.
buffer	is a pointer to a buffer holding a zero-terminated string.
len	is the length of the buffer.

Return

This function returns the number of characters actually written. This is `len` unless an error occurs.

4.11.8 Hostif_DebugPause

This function waits for the user to press any key.

Syntax

```
void Hostif_DebugPause(const struct RDI_HostosInterface *hostif)
```

where:

hostif	is the handle for the host interface.

4.12 Tracer

This section describes the functions provided by the tracer module, `tracer.c`.

Note

These functions are not exported. If you want to use any of these functions in your model, you must build your model together with `tracer.c`.

The default implementations of these functions can be changed by compiling `tracer.c` with `EXTERNAL_DISPATCH` defined.

The formats of `Trace_State` and `Trace_Packet` are documented in `tracer.h`.

4.12.1 Tracer_Open

This function is called when the tracer is initialized.

Syntax

```
unsigned Tracer_Open(Trace_State *ts)
```

Usage

The implementation in `tracer.c` opens the output file from this function, and writes a header.

4.12.2 Tracer_Dispatch

This function is called on each traced event for every instruction, event, or memory access.

Syntax

```
void Tracer_Dispatch(Trace_State *ts, Trace_Packet *packet)
```

Usage

In `tracer.c`, this function writes the packet to the trace file.

4.12.3 Tracer_Close

This function is called at the end of tracing.

Syntax

```
void Tracer_Close(Trace_State *ts)
```

Usage

The file `tracer.c` uses this to close the trace file.

4.12.4 Tracer_Flush

This function is called when tracing is disabled.

Syntax

```
extern void Tracer_Flush(Trace_State *ts)
```

Usage

The file `tracer.c` uses this to flush output to the trace file.

4.13 Map files

The type and speed of memory in a simulated system can be detailed in a map file. A map file defines the number of regions of attached memory, and for each region:

- the address range to which that region is mapped
- the data bus width in bytes
- the access time for the memory region.

`armsd` expects the map file to be called `armsd.map`, in the current working directory.

AXD and ADW/ADU accept map files of any name, provided that they have the extension `.map`. See *ADS Debuggers Guide* for details of how to use a particular map file in a debugging session.

To calculate the number of wait states for each possible type of memory access, the ARMulator uses the access times supplied in the map file, and the clock frequency from the debugger (see *ADS Debuggers Guide*).

Note

A memory map file defines the characteristics of the memory areas defined in `peripherals.ami` (see *ARMulator configuration files* on page 4-57). A `.map` file must define rw areas that are at least as large as those specified for the heap and stack in `peripherals.ami`, and at the same locations. If this is not the case, Data Aborts are likely to occur during execution.

4.13.1 Format of a map file

The format of each line is:

`start size name width access{*} read-times write-times`

where:

`start`	is the start address of the memory region in hexadecimal, for example `80000`.
`size`	is the size of the memory region in hexadecimal, for example, `4000`.
`name`	is a single word that you can use to identify the memory region when memory access statistics are displayed. You can use any name. To ease readability of the memory access statistics, give a descriptive name such as SRAM, DRAM, or EPROM.
`width`	is the width of the data bus in bytes (that is, 1 for an 8-bit bus, 2 for a 16-bit bus, or 4 for a 32-bit bus).

`access`	describes the type of accesses that can be performed on this region of memory:

`r`	for read-only.
`w`	for write-only.
`rw`	for read-write.
`-`	for no access. Any access causes a Data or Prefetch Abort.

An asterisk (*) can be appended to *`access`* to describe a Thumb-based system that uses a 32-bit data bus to memory, but which has a 16-bit latch to latch the upper 16 bits of data, so that a subsequent 16-bit sequential access can be fetched directly out of the latch.

`read-times`

describes the nonsequential and sequential read times in nanoseconds. These must be entered as the nonsequential read access time followed by a slash (`/`), followed by the sequential read access time. Omitting the slash and using only one figure indicates that the nonsequential and sequential access times are the same.

Note

For accurate modelling of real devices, you might have to add a signal propagation delay (20 to 30ns) to the read and write times quoted for a memory chip.

`write-times`

describes the nonsequential and sequential write times. The format is the same as that given for read times.

The following examples assume a clock speed of 20MHz.

Example 1

```
0 80000000 RAM 4 rw 135/85 135/85
```

This describes a system with a single continuous section of RAM from `0` to `0x7FFFFFFF` with a 32-bit data bus, read-write access, nonsequential access time of 135ns, and sequential access time of 85ns.

Example 2

This example describes a typical embedded system with 32KB of on-chip memory, 16-bit ROM and 32KB of external DRAM:

```
00000000 8000 SRAM  4 rw   1/1     1/1
00008000 8000 ROM   2 r  100/100 100/100
00010000 8000 DRAM  2 rw 150/100 150/100
7FFF8000 8000 Stack 2 rw 150/100 150/100
```

There are four regions of memory:

- A fast region from `0` to `0x7FFF` with a 32-bit data bus. This is labeled SRAM.
- A slower region from `0x8000` to `0xFFFF` with a 16-bit data bus. This is labelled ROM and contains the image code. It is marked as read-only.
- A region of RAM from `0x10000` to `0x17FFF` that is used for image data.
- A region of RAM from `0x7FFF8000` to `0x7FFFFFFF` that is used for stack data. The stack pointer is initialized to `0x80000000`.

In the final hardware, the two distinct regions of the external DRAM are combined. This does not make any difference to the accuracy of the simulation.

To represent fast (no wait state) memory, the SRAM region is given access times of 1ns. In effect, this means that each access takes 1 clock cycle, because ARMulator rounds this up to the nearest clock cycle. However, specifying it as 1ns allows the same map file to be used for a number of simulations with differing clock speeds.

——— Note ———

To ensure accurate simulations, make sure that all areas of memory likely to be accessed by the image you are simulating are described in the memory map.

———————————

To ensure that you have described all areas of memory that you think the image accesses, you can define a single memory region that covers the entire address range as the last line of the map file. For example, you could add the following line to the above description:

```
00000000 80000000 Dummy 4 - 1/1 1/1
```

You can then detect if any reads or writes are occurring outside the regions of memory you expect using the `print $memory_statistics` command.

——— Note ———

A dummy memory region must be the *last* entry in a map file.

———————————

Reading the memory statistics

To read the memory statistics use the command:

```
print $memory_statistics
```

This reports the statistics in the following form:

Example 4-1

```
address  name    W acc R(N/S)  W(N/S)    reads(N/S)   writes(N/S)  time (ns)
00000000 Dummy   4 -     1/1     1/1        0/0          0/0      0
7FFF8000 Stack   2 rw  150/100 150/100   9290/10590    4542/11688  8538300
00010000 DRAM    2 rw  150/100 150/100  18817/18      11031/140    8915800
00008000 ROM     2 r   100/100 100/100  48638/176292      0/0      44817000
00000000 SRAM    4 rw    1/1     1/1        0/0          0/0     0
```

`print $memstats` is a short version of `print $memory_statistics`.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.14 ARMulator configuration files

This section contains the following subsections:

- *Predefined tags* on page 4-57
- *Processors* on page 4-58.

ARMulator configuration files (`.ami` files) are ToolConf files. See *ToolConf* on page 4-60.

Depending on your system, these are be located in one of:

- `*install_directory*\bin`
- `*install_directory*/solaris/bin`
- `*install_directory*/hpux/bin`.

You can edit `.ami` files, or add your own.

By default, there are the following `.ami` files, all in `ADSv1_1\Bin`:

- `default.ami`
- `peripherals.ami`
- `example1.ami`

ARMulator loads all `.ami` files it finds on any of the paths it finds in the environment variable `armconf`. This is initially set up to point to `*install_directory*\Bin`.

4.14.1 Predefined tags

Before reading `.ami` files, ARMulator creates several tags itself, based on the settings you give to the debugger. These are given in Table 4-7. Preprocessing directives in `.ami` files use these tags to control the configuration.

Table 4-7 Tags predefined by ARMulator

Tag	Description
`CPUSpeed`	Set to the speed set in the configuration window of AXD, ADU or ADW, or in the `-clock` command line option for `armsd`. For example, `CPUSpeed=30MHz`.
`MCLK` and `FCLK`	Set to the same value as `CPUSpeed`, if that value is not zero. Not set if `CPUSpeed` is zero.
`ByteSex`	Set to `L` or `B` if a bytesex is specified from the debugger. Not set otherwise.
`FPE`	Set to `True` or `False` from the debugger.

4.14.2 Processors

The processors region is a child ToolConf database (see *ToolConf* on page 4-60). It has a full list of processors supported by the ARMulator. This list is the basis of the list of processors in AXD, ADU and ADW, and the list of accepted arguments for the `-processor` option of `armsd`.

You can add a variant processor to this list, for example to include a particular memory model in the definition. See `install_directory\Bin\example1.ami` for examples.

`Default` specifies the processor to use if no other processor is specified. Each other entry in the `Processors` region is the name of a processor.

Example 4-2 declares two processors, TRACED_ARM10 and PROFILED_ARM7.

Example 4-2 Processors in a toolconf file

```
{Processors

{TRACED_ARM10=ARM10200E

;;CPUSPEED=400MHz

;Memory clock divisor.
;(The AHB runs this many times slower than the core.)
MCCFG=4

{Tracer=Default_Tracer
;; Output options - can be plaintext to file, binary to file or to RDI log
;; window. (Checked in the order RDILog, File, BinFile.)
RDILog=False
File=armul.trc
BinFile=armul.trc
;; Tracer options - what to trace
TraceInstructions=True
TraceRegisters=False
TraceMemory=True
TraceEvents=False
;; Flags - disassemble instructions; start up with tracing enabled;
Disassemble=True
StartOn=True
}
;End TRACED_ARM10
}

{PROFILED_ARM7=ARM720T
{Profiler=Default_Profiler
}
}

;End Processors
}
```

Finding the configuration for a selected processor

ARMulator uses the following algorithm to find a configuration for a selected processor:

1. Set the current region to be `Processors`.
2. Find the selected processor in the current region.
3. If the tag has a child, that child is the required configuration.

Adding a variant processor model

Suppose you have created a memory model called `MyASIC`, designed to be combined with an ARM7TDMI® processor core to make a new microcontroller called `ARM7TASIC`. To allow this to be selected from AXD, ADW, ADU or `armsd`, add a `.ami` file modeled on `example1.ami`.

4.15 ToolConf

This section contains the following subsections:

4.15.1 Toolconf overview

ToolConf is a module within ARMulator. A ToolConf file is a tree-structured database consisting of tag and value pairs. Tags and values are strings, and are usually case-insensitive.

You can find a value associated with a tag from a ToolConf database, or add or change a value.

If a tag is given a value more than once, the first value is used.

4.15.2 File format

The following are typical ToolConf database lines:

```
TagA=ValueA
TagA=NewValue
Othertag
Othertag=Othervalue
;; Lines starting with ; (semicolon) are comments.
; Tag=Value
```

The first line creates a tag in the ToolConf called `TagA`, with value `ValueA`.

The second line has no effect, as `TagA` already has a value.

The third line creates a tag called `Othertag`, with no value.

The fourth line gives the value `Othervalue` to `Othertag`.

There must be no whitespace at the beginning of database lines, in tags, in values, or between tags or values and the = symbol.

Conventionally, ordinary comments start with two semicolons. Lines starting with one semicolon are usually commented-out lines. You can comment out a line to disable it, or uncomment a commented-out line to enable it.

A comment must be on a line by itself.

File header

If you add any toolconf files, the first line of the file must be:

```
;; ARMulator configuration file type 3
```

ARMulator ignores any `.ami` or `.dsc` files that do not begin with this header.

Tree structure

Each tag can have another ToolConf database associated with it, called its child. When a tag lookup is performed on a child, if the tag is not found in the child, the search continues in the parent, and if necessary in the parent's parent and so on until the tag is found.

This means that the child only includes tags whose values are different from those of the same tag in the parent.

If child databases are specified more than once for the same parent, the child databases are merged.

Specifying children

There are two ways of specifying children in a ToolConf database.

One is more suited to specifying large children:

```
{ TagP=ValueP
TagC1=ValueC1
TagC2=ValueC2
}
```

This creates a tag called `TagP`, with the value `ValueP`, and a child database. Two tags are given values in the child.

The other is more suited to specifying small children:

```
TagP:TagC=ValueC
```

This creates a tag called `TagP`, with no `value.` `TagP` has a child in which one tag is created, `TagC`, with value `ValueC`. It is equivalent to:

```
{ TagP
TagC=ValueC
}
```

Conditional expressions

The full `#if...#elif...#else...#endif` syntax is supported. You can use this to skip regions of a ToolConf database. Expressions use tags from the file, for example, the C preprocessor sequence:

```
#define Control True

#if defined(Control) && Control==True
#define controlIsTrue Yes
#endif
```

maps to the ToolConf sequence:

```
Control=True

#if Control && Control=True
ControlIsTrue=Yes
#endif
```

A condition is evaluated from left to right, on the contents of the configuration at that point. Table 4-8 shows the operators that can be used in ToolConf conditional expressions.

Table 4-8 Operators in ToolConf preprocessor expressions

Operator	Example	Description
none	`Tag`	Test for existence of tag definition
`==`	`Tag==Value`	Case-insensitive string equality test
`!=`	`Tag!=Value`	Case-insensitive string inequality test
`(...)`	`(Tag==Value)`	Grouping
`&&`	`TagA==ValueA && TagB==ValueB`	Boolean AND
`||`	`TagA==ValueA || TagB==ValueB`	Boolean OR
`!`	`!(Tag==Value)`	Boolean NOT

File inclusion

You can use the `#include` directive to include one ToolConf file in another. The directive is ignored if it is in a region which is being skipped under control of a conditional expression.

4.15.3 Boolean flags in a ToolConf database

Table 4-9 shows the full set of permissible values for Boolean flags. The strings are case-insensitive.

Table 4-9 Boolean values

True	False
True	False
On	Off
High	Low
Hi	Lo
1	0
T	F

4.15.4 SI units in a ToolConf database

Some values can be specified using SI (Système Internationale) units, for example:

```
ClockSpeed=10MHz
MemorySize=2Gb
```

The scaling factor is set by the prefix to the unit. ARMulator only accepts k, M, or G prefixes for kilo, mega, and giga. These correspond to scalings of 10^3, 10^6, and 10^9, or 2^{10}, 2^{20}, and 2^{30}. ARMulator decides which scaling to use according to context.

4.15.5 ToolConf_Lookup

This function performs a lookup on a specified tag in an `.ami` or `.dsc` file. If the tag is found, its associated value is returned. Otherwise, `NULL` is returned.

Syntax

```
const char *ToolConf_Lookup(toolconf hashv, tag_t tag)
```

where:

hashv	is the database to perform the lookup on.
tag	is the tag to search for in the database. The tag is case-dependent.

Return

The function returns:

- a **`const`** pointer to the tag value, if the search is successful
- `NULL`, if the search is not successful.

Example

```
const char *option = ToolConf_Lookup(db, ARMulCnf_Size);

/* ARMulCnf_Size is defined in armcnf.h */
```

4.15.6 ToolConf_Cmp

This function performs a case-insensitive comparison of two ToolConf database tag values.

Syntax

```
int ToolConf_Cmp(const char *s1, const char *s2)
```

where:

s1	is a pointer to the first string value to compare.
s2	is a pointer to the second string value to compare.

Return

The function returns:

- 1, if the strings are identical
- 0, if the strings are different.

Example

```
if (ToolConf_Cmp(option, "8192"))
```

4.16 Reference peripherals

Two reference peripherals are detailed here:

- *Interrupt controller* on page 4-67
- *Timer* on page 4-69.

4.16.1 Interrupt controller

The base address of the interrupt controller, `IntBase`, is configurable (see *Interrupt controller* on page 2-24).

Table 4-10 shows the location of individual registers.

Table 4-10 Interrupt controller memory map

Address	Read	Write
`IntBase`	IRQStatus	Reserved
`IntBase + 004`	IRQRawStatus	Reserved
`IntBase + 008`	IRQEnable	IRQEnableSet
`IntBase + 00C`	Reserved	IRQEnableClear
`IntBase + 010`	Reserved	IRQSoft
`IntBase + 100`	FIQStatus	Reserved
`IntBase + 104`	FIQRawStatus	Reserved
`IntBase + 108`	FIQEnable	FIQEnableSet
`IntBase + 10C`	Reserved	FIQEnableClear

Interrupt controller defined bits

The FIQ interrupt controller is one bit wide. It is located on bit 0.

Table 4-11 gives details of the interrupt sources associated with bits 1 to 5 in the IRQ interrupt controller registers. You can use bit 0 for a duplicate FIQ input.

Table 4-11 Interrupt sources

Bit	Interrupt source
0	FIQ source
1	Programmed interrupt
2	Communications channel Rx
3	Communications channel Tx
4	Timer 1
5	Timer 2

Note

Timer 1 and Timer 2 can be configured to use different bits in the IRQ controller registers, see *Timer* on page 2-24.

4.16.2 Timer

The base address of the timer, `TimerBase`, is configurable (see *Timer* on page 2-24).

See Table 4-12 for the location of individual registers.

Table 4-12 Timer memory map

Address	Read	Write
`TimerBase`	Timer1Load	Timer1Load
`TimerBase + 04`	Timer1Value	Reserved
`TimerBase + 08`	Timer1Control	Timer1Control
`TimerBase + 0C`	Reserved	Timer1Clear
`TimerBase + 10`	Reserved	Reserved
`TimerBase + 20`	Timer2Load	Timer2Load
`TimerBase + 24`	Timer2Value	Reserved
`TimerBase + 28`	Timer2Control	Timer2Control
`TimerBase + 2C`	Reserved	Timer2Clear
`TimerBase + 30`	Reserved	Reserved

Timer load registers

Write a value to one of these registers to set the initial value of the corresponding timer counter. You must write the top 16 bits as zeroes.

If the timer is in periodic mode, this value is also reloaded to the timer counter when the counter reaches zero.

If you read from this register, the bottom 16 bits return the value that you wrote. The top 16 bits are undefined.

Timer value registers

Timer value registers are read-only. The bottom 16 bits give the current value of the timer counter. The top 16 bits are undefined.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Timer clear registers

Timer clear registers are write-only. Writing to one of them clears an interrupt generated by the corresponding timer.

Timer control registers

See Table 4-14 and Table 4-13 for details of timer register bits. Only bits 7, 6, 3, and 2 are used. You must write all others as zeroes.

Table 4-13 Clock prescaling using bits 2 and 3

Bit 3	Bit 2	Clock divided by	Stages of prescale
0	0	1	0
0	1	16	4
1	0	256	8
1	1	Undefined	-

The counter counts downwards. It counts **BCLK** cycles, or **BCLK** cycles divided by 16 or 256. Bits 2 and 3 define the prescaling applied to the clock.

Table 4-14 Timer enable and mode control using bits 6 and 7

	0	1
Bit 7	Timer disabled	Timer enabled
Bit 6	Free-running mode	Periodic mode

In free-running mode, the timer counter overflows when it reaches zero, and continues to count down from `0xFFFF`.

In periodic mode, the timer generates an interrupt when the counter reaches zero. It then reloads the value from the load register and continues to count down from this value.

Chapter 5
Semihosting

This chapter describes the semihosting mechanism. Semihosting provides code running on an ARM target use of facilities on a host computer that is running an ARM debugger. Examples of such facilities include the keyboard input, screen output, and disk I/O. This chapter contains the following sections:

Copyright © 1999, 2000 ARM Limited. All rights reserved.

5.1 Semihosting

Semihosting is a mechanism for ARM targets to communicate input/output requests from application code to a host computer running a debugger. This mechanism could be used, for example, to allow functions in the C library, such as `printf()` and `scanf()`, to use the screen and keyboard of the host rather than having a screen and keyboard on the target system.

This is useful because development hardware often does not have all the input and output facilities of the final system. Semihosting allows the host computer to provide these facilities.

Semihosting is implemented by a set of defined *software interrupt* (*SWI*) operations. The application invokes the appropriate SWI and the debug agent then handles the SWI exception. The debug agent provides the required communication with the host.

In many cases, the semihosting SWI will be invoked by code within library functions. The application can also invoke the semihosting SWI directly. Refer to the C library descriptions in the *ADS Compiler, Linker, and Utilities Guide* for more information on support for semihosting in the ARM C library.

Figure 5-1 on page 5-2 shows an overview of semihosting.

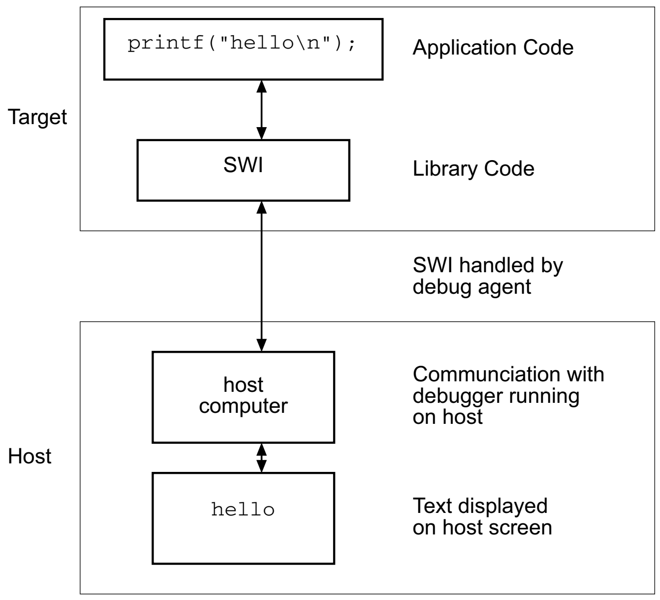
printf("hello\n");
Application Code
Target
SWI
Library Code
SWI handled by debug agent
host computer
Communciation with debugger running on host
Host
hello
Text displayed on host screen

Figure 5-1 Semihosting overview

The semihosting SWI interface is common across all debug agents provided by ARM. Semihosted operations will work under ARMulator, RealMonitor, Angel, or Multi-ICE without any requirement for porting.

For further information on semihosting and the C libraries, see the *C and C++ Libraries* chapter in *ADS Compiler, Linker, and Utilities Guide*. See also the *Writing Code for ROM* chapter in *ADS Developer Guide*.

5.1.1 The SWI interface

The ARM and Thumb SWI instructions contain a field that encodes the SWI number used by the application code. This number can be decoded by the SWI handler in the system. See the chapter on exception handling in *ADS Developer Guide* for more information on SWI handlers.

Semihosting operations are requested using a single SWI number. This leaves the other SWI numbers available for use by the application or operating system. The SWI used for semihosting is:

`0x123456`	in ARM state
`0xAB`	in Thumb state

The SWI number indicates to the debug agent that the SWI is a semihosting request. In order to distinguish between operations, the operation type is passed in r0, rather than being encoded in the SWI number. All other parameters are passed in a block that is pointed to by r1.

The result is returned in r0, either as an explicit return value or as a pointer to a data block. Even if no result is returned, assume that r0 is corrupted. The application must preserve registers r1-r3 (and r0 if used) when a semihosting request is made.

The available semihosting operation numbers passed in r0 are allocated as follows:

`0x00` **to** `0x31`	These are used by ARM.
`0x32` **to** `0xFF`	These are reserved for future use by ARM.
`0x100` **to** `0x1FF`	These are reserved for user applications. They will not be used by ARM. If you are writing your own SWI operations, however, you must use a different SWI number rather than using the semihosted SWI number and these operation type numbers.
`0x200` **to** `0xFFFFFF`	These are undefined. They are not currently used and not recommended for use.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

In the following sections, the number in parentheses after the operation name is the value placed into r0. For example `SYS_OPEN` (`0x01`).

If you are calling SWIs from assembly language code it is best to use the operation names that are defined in `ARMulate\armulif\semihost.h`. You can define the operation names with an `EQU` directive. For example:

```
SYS_OPEN    EQU 0x01
SYS_CLOSE   EQU 0x02
```

Changing the semihosting SWI numbers

It is strongly recommended that you do not change the semihosting SWI numbers `0x123456` (ARM) or `0xAB` (Thumb). To do so you must:

- change all the code in your system, including library code, to use the new SWI number
- reconfigure your debugger to use the new SWI number.

5.2 Semihosting implementation

The functionality provided by semihosting is basically the same on all debug hosts. The implementation of semihosting, however, differs between hosts.

5.2.1 ARMulator

When a semihosting SWI is encountered, the semihosted functionality within ARMulator is automatically invoked. ARMulator traps the SWI directly and the instruction in the SWI entry in the vector table is not executed.

To turn the support for semihosting off in ARMulator, change `default_semihost` in the `default.ami` file to `no_semihost`.

See the *Peripheral models* on page 2-23 for more details.

5.2.2 RealMonitor

RealMonitor implements a SWI handler that must be integrated with your system to enable semihosting support.

When the target executes a semihosted SWI instruction, the Realmonitor SWI handler carries out the required communication with the host.

For further information refer to the documentation supplied with RealMonitor.

5.2.3 Angel

The Angel debug monitor installs a SWI handler during its initialization. This occurs when the target powers up.

When the target executes a semihosted SWI instruction, the Angel SWI handler carries out the required communication with the host.

5.2.4 Multi-ICE

When using Multi-ICE in default configuration, semihosting is implemented as follows:

1. On ARM7 processors:
   a. A breakpoint is set on the SWI vector.
   b. When this breakpoint is hit, Multi-ICE examines the SWI number.
   c. If the SWI is recognized as a semihosting SWI, Multi-ICE emulates it and transparently restarts execution of the application.

      If the SWI is not recognized as a semihosting SWI, Multi-ICE halts the processor and reports an error.

2. On other processors:
   a. Vector-catch logic traps SWIs.
   b. If the SWI is recognized as a semihosting SWI, Multi-ICE emulates it and transparently restarts execution of the application.

      If the SWI is not recognized as a semihosting SWI, Multi-ICE halts the processor and reports an error.

This semihosting mechanism can be disabled or changed by the following debugger internal variables:

`$semihosting_enabled`

Set this variable to 0 to disable semihosting. If you are debugging an application running from ROM, this allows you to use an additional watchpoint unit.

Set this variable to 1 to enable semihosting. This is the default.

Set this variable to 2 to enable Debug Communications Channel semihosting.

You can only set the S bit in `$vector_catch` if semihosting is disabled.

`$semihosting_vector`

This variable controls the location of the breakpoint set by Multi-ICE to detect a semihosted SWI. It is set to the SWI entry in the exception vector table (`0x8`) by default.

If your application requires semihosting as well as having its own SWI handler, set `$semihosting_vector` to an address in your SWI handler. This address must point to an instruction that is only executed if your SWI handler has identified a call to a semihosting SWI. All registers must already have been restored to whatever values they had on entry to your SWI handler.

Multi-ICE handles the semihosted SWI and then examines the contents of lr and returns to the instruction following the SWI instruction in your code.

Regardless of the value of `$vector_catch`, all exceptions and interrupts are trapped and reported as an error condition.

For details of how to modify debugger internal variables, see the appropriate debugger documentation.

5.2.5 Multi-ICE DCC semihosting

Multi-ICE can also use the debug communications channel so that the core is not stopped while semihosting takes place. This is enabled by setting `$semihosting_enabled` to 2. Refer to the *Multi-ICE User Guide* [ARM DUI 0048] for more details.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

5.3 Adding an application SWI handler

It can be useful to have both the semihosted SWIs and your own application-specific SWIs available. In such cases you must ensure that the two SWI mechanisms cooperate correctly. The way to ensure this depends upon the debug agent in use.

5.3.1 ARMulator

To get your own handler and the semihosting handler to cooperate, simply install your SWI handler into the SWI entry in the vector table. No other actions are required.

When an appropriate SWI is reached in your code, ARMulator detects that it is not a semihosting SWI and executes the instruction in the SWI entry of the vector table instead. This instruction must branch to your own SWI handler.

5.3.2 RealMonitor

The RealMonitor SWI handler must be integrated with your application to enable semihosting (see the documentation supplied with RealMonitor).

5.3.3 Angel

Application SWI handlers are added by:

1. Saving the SWI vector (as installed by Angel).
2. Adjusting the contents of the SWI vector to point to the application SWI handler. (This is called *chaining*.) This is described in more detail in the exception handling section of the *ADS Developer Guide*.

5.3.4 Multi-ICE

To ensure that the application SWI handler will successfully cooperate with Multi-ICE semihosting mechanism:

1. Install the application SWI handler into the vector table.
2. Modify `$semihosting_vector` to point to a location at the end of the application handler. This point in the handler must only be reached if your handler does not handle the SWI.

Before Multi-ICE traps the SWI, your SWI handler must restore all registers to the values they had when your SWI handler was entered. Typically, this means that your SWI handler must store the registers to a stack on entry and restore them before falling through to the semihosting vector address.

Caution

It is essential that the actual position `$semihosting_vector` points to within the application handler is correct.

See exception handling in the *ADS Developer Guide* for writing SWI handlers.

The following example SWI handler can detect if it fails to handle a SWI. In this case, it branches to an error handler:

```
; r0 = 1 if SWI handled
    CMP r0, #1                ; Test if SWI has been handled.
    BNE NoSuchSWI             ; Call unknown SWI handler.
    LDMFD sp!, {r0}           ; Unstack SPSR...
    MSR spsr_cxsf, r0         ; ...and restore it.
    LDMFD sp!, {r0-r12,pc}^   ; Restore registers and return.
```

This code could be modified to co-operate with Multi-ICE semihosting as follows:

```
; r0 = 1 if SWI handled
    CMP r0, #1                ; Test if SWI has been handled.
    LDMFD sp!, {r0}           ; Unstack SPSR...
    MSR spsr_cxsf, r0         ; ...and restore it.
    LDMFD sp!, {r0-r12,lr}    ; Restore registers.
    MOVEQS pc, lr             ; Return if SWI handled.
Semi_SWI
    MOVS pc,lr                ; Fall through to Multi-ICE
                              ; interface handler.
```

The `$semihosting_vector` variable must be set up to point to the address of `Semi_SWI`. The instruction at `Semi_SWI` never gets executed because Multi-ICE returns directly to the application after processing the semihosted SWI (see Figure 5-2 on page 5-10).

Caution

Using a normal SWI return instruction ensures that the application does not crash if the semihosting breakpoint is not set up. The semihosting action requested is not carried out and the handler simply returns.

You must also be careful if you modify `$semihosting_vector` to point to the fall-through part of the application SWI handler. If `$semihosting_vector` changes value before the application starts execution, and semihosted SWIs are invoked before the application SWI handler is installed, an unknown watchpoint error will occur.

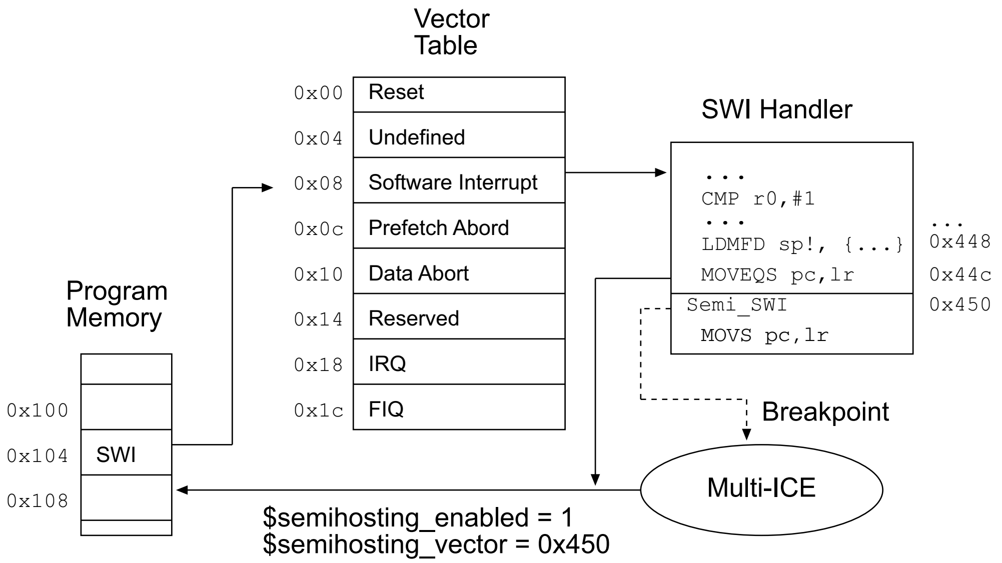

Figure 5-2 Semihosting with breakpoint

The error occurs because the vector table location for the SWI has not yet had the application handler installed into it and might still contain the software breakpoint bit pattern. Because the `$semihosting_vector` address has moved to a place that cannot currently be reached, Multi-ICE no longer knows about the triggered breakpoint. To prevent this from happening, you must change the contents of `$semihosting_vector` only at the point in your code where the application SWI handler is installed into the vector table.

Note

If semihosting is not required at all by an application, this process can be simplified by setting `$semihosting_enabled` to 0.

5.3.5 Multi-ICE DCC semihosting

When using the DCC semihosting mechanism, adding an application SWI handler must be done in exactly the same way as non-DCC semihosting (see *Multi-ICE* on page 5-8).

5.4 Semihosting SWIs

The SWIs listed in Table 5-1 implement the semihosted operations. These operations are used by C library functions such as `printf()` and `scanf()`. They can be treated as ATPCS function calls. However, except for r0 that contains the return status, they restore the registers they are called with before returning.

Some targets provide additional semihosting calls. See the *ARM Firmware Suite* (AFS) documentation for details of SWIs provided by AFS.

Table 5-1 Semihosting SWIs

SWI	Description
SYS_OPEN (0x01) on page 5-12	Open a file on the host
SYS_CLOSE (0x02) on page 5-14	Close a file on the host
SYS_WRITEC (0x03) on page 5-14	Write a character to the console
SYS_WRITE0 (0x04) on page 5-14	Write a null-terminated string to the console
SYS_WRITE (0x05) on page 5-15	Write to a file on the host
SYS_READ (0x06) on page 5-16	Read the contents of a file into a buffer
SYS_READC (0x07) on page 5-17	Read a byte from the console
SYS_ISERROR (0x08) on page 5-17	Determine if a return code is an error
SYS_ISTTY (0x09) on page 5-18	Check whether a file is connected to an interactive device
SYS_SEEK (0x0A) on page 5-18	Seek to a position in a file
SYS_FLEN (0x0C) on page 5-19	Return the length of a file
SYS_TMPNAM (0x0D) on page 5-19	Return a temporary name for a file
SYS_REMOVE (0x0E) on page 5-20	Remove a file from the host
SYS_RENAME (0x0F) on page 5-20	Rename a file on the host
SYS_CLOCK (0x10) on page 5-21	Number of centiseconds since execution started
SYS_TIME (0x11) on page 5-21	Number of seconds since January 1, 1970
SYS_SYSTEM (0x12) on page 5-22	Pass a command to the host command-line interpreter
SYS_ERRNO (0x13) on page 5-23	Get the value of the C library `errno` variable

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Table 5-1 Semihosting SWIs (continued)

SWI	Description
SYS_GET_CMDLINE (0x15) on page 5-24	Get the command-line used to call the executable
SYS_HEAPINFO (0x16) on page 5-25	Get the system heap parameters
SYS_ELAPSED (0x30) on page 5-26	Get the number of target ticks since execution started
SYS_TICKFREQ (0x31) on page 5-26	Determine the tick frequency

Note

When used with Angel, these SWIs use the serializer and the global register block, and they can take a significant length of time to process.

5.4.1 SYS_OPEN (0x01)

Open a file on the host system. The file path is specified either as relative to the current directory of the host process, or absolutely, using the path conventions of the host operating system.

The ARM targets interpret the special path name `:tt` as meaning the console input stream (for an open-read) or the console output stream (for an open-write). Opening these streams is performed as part of the standard startup code for those applications that reference the C stdio streams.

Entry

On entry, r1 contains a pointer to a three-word argument block:

word 1 This is a pointer to a null-terminated string containing a file or device name.

word 2 This is an integer that specifies the file opening mode. Table 5-2 gives the valid values for the integer, and their corresponding ANSI C `fopen()` mode.

word 3 This is an integer that gives the length of the string pointed to by word 1. The length does not include the terminating null character that must be present.

Table 5-2 Value of mode

mode	0	1	2	3	4	5	6	7	8	9	10	11
ANSI C fopen mode	r	rb	r+	r+b	w	wb	w+	w+b	a	ab	a+	a+b

Return

On exit, r0 contains:

- a nonzero handle if the call is successful
- –1 if the call is not successful.

5.4.2 SYS_CLOSE (0x02)

Closes a file on the host system. The handle must reference a file that was opened with SYS_OPEN.

Entry

On entry, r1 contains a pointer to a one-word argument block:

word 1	This is a file handle referring to an open file.

Return

On exit, r0 contains:

- 0 if the call is successful
- –1 if the call is not successful.

5.4.3 SYS_WRITEC (0x03)

Writes a character byte, pointed to by r1, to the debug channel. When executed under an ARM debugger, the character appears on the display device connected to the debugger.

Entry

On entry, r1 contains a pointer to the character.

Return

None. Register r0 is corrupted.

5.4.4 SYS_WRITE0 (0x04)

Writes a null-terminated string to the debug channel. When executed under an ARM debugger, the characters appear on the display device connected to the debugger.

Entry

On entry, r1 contains a pointer to the first byte of the string.

Return

None. Register r0 is corrupted.

5.4.5 SYS_WRITE (0x05)

Writes the contents of a buffer to a specified file at the current file position. The file position is specified either:

- explicitly, by a SYS_SEEK
- implicitly as one byte beyond the previous SYS_READ or SYS_WRITE request.

The file position is at the start of the file when the file is opened, and is lost when the file is closed.

Perform the file operation as a single action whenever possible. For example, do not split a write of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, r1 contains a pointer to a three-word data block:

word 1	This contains a handle for a file previously opened with SYS_OPEN
word 2	This points to the memory containing the data to be written
word 3	This contains the number of bytes to be written from the buffer to the file.

Return

On exit, r0 contains:

- 0 if the call is successful
- the number of bytes that are not written, if there is an error.

5.4.6 SYS_READ (0x06)

Reads the contents of a file into a buffer. The file position is specified either:

- explicitly by a SYS_SEEK
- implicitly one byte beyond the previous SYS_READ or SYS_WRITE request.

The file position is at the start of the file when the file is opened, and is lost when the file is closed. Perform the file operation as a single action whenever possible.For example, do not split a read of 16KB into four 4KB chunks unless there is no alternative.

Entry

On entry, r1 contains a pointer to a four-word data block:

word 1	This contains a handle for a file previously opened with SYS_OPEN.
word 2	This points to a buffer.
word 3	This contains the number of bytes to read to the buffer from the file.
word 4	This is an integer that specifies the file mode. Table 5-2 on page 5-13 gives the valid values for the integer, and their corresponding ANSI C `fopen()` modes.

Return

On exit, r0 contains:

- 0 if the call is successful
- the number of bytes not read, if there is an error.

If the handle is for an interactive device (that is, SYS_ISTTY returns –1 for this handle), a nonzero return from SYS_READ indicates that the line read did not fill the buffer.

5.4.7 SYS_READC (0x07)

Reads a byte from the console.

Entry

Register r1 must contain zero. There are no other parameters or values possible.

Return

On exit, r0 contains the byte read from the console.

5.4.8 SYS_ISERROR (0x08)

Determines whether the return code from another semihosting call is an error status or not. This call is passed a parameter block containing the error code to examine.

Entry

On entry, r1 contains a pointer to a one-word data block:

word 1	This is the required status word to check.

Return

On exit, r0 contains:

- 0 if the status word is not an error indication
- a nonzero value if the status word is an error indication.

5.4.9 SYS_ISTTY (0x09)

Checks whether a file is connected to an interactive device.

Entry

On entry, r1 contains a pointer to a one-word argument block:

word 1	This is a handle for a previously opened file object.

Return

On exit, r0 contains:

- 1 if the handle identifies an interactive device
- 0 if the handle identifies a file
- a value other than 1 or 0 if an error occurs.

5.4.10 SYS_SEEK (0x0A)

Seeks to a specified position in a file using an offset specified from the start of the file. The file is assumed to be a byte array and the offset is given in bytes.

Entry

On entry, r1 contains a pointer to a two-word data block:

word 1	This is a handle for a seekable file object.
word 2	This is the absolute byte position to be sought to.

Return

On exit, r0 contains:

- 0 if the request is successful
- A negative value if the request is not successful. SYS_ERRNO can be used to read the value of the host `errno` variable describing the error.

——— **Note** ———

The effect of seeking outside the current extent of the file object is undefined.

————————————

5.4.11 SYS_FLEN (0x0C)

Returns the length of a specified file.

Entry

On entry, r1 contains a pointer to a one-word argument block:

word 1	This is a handle for a previously opened, seekable file object.

Return

On exit, r0 contains:

- the current length of the file object, if the call is successful
- –1 if an error occurs.

5.4.12 SYS_TMPNAM (0x0D)

Returns a temporary name for a file identified by a system file identifier.

Entry

On entry, r1 contains a pointer to a three-word argument block:

word 1	This is a pointer to a buffer.
word 2	This is a target identifier for this filename. Its value must be an integer in the range 0 to 255.
word 3	This contains the length of the buffer. The length must be at least the value of `L_tmpnam` on the host system.

Return

On exit, r0 contains:

- 0 if the call is successful
- –1 if an error occurs.

The buffer pointed to by r1 contains the filename.

If you use the same target identifier again, the same filename is returned.

5.4.13 SYS_REMOVE (0x0E)

Caution

Deletes a specified file on the host filing system.

Entry

On entry, r1 contains a pointer to a two-word argument block:

word 1	This points to a null-terminated string that gives the pathname of the file to be deleted.
word 2	This is the length of the string.

Return

On exit, r0 contains:

- 0 if the delete is successful
- a nonzero, host-specific error code if the delete fails.

5.4.14 SYS_RENAME (0x0F)

Renames a specified file.

Entry

On entry, r1 contains a pointer to a four-word data block:

word 1	This is a pointer to the name of the old file.
word 2	This is the length of the old file name.
word 3	This is a pointer to the new file name.
word 4	This is the length of the new file name.

Both strings are null-terminated.

Return

On exit, r0 contains:

- 0 if the rename is successful
- a nonzero, host-specific error code if the rename fails.

5.4.15 SYS_CLOCK (0x10)

Returns the number of centiseconds since the execution started.

Values returned by this SWI can be of limited use for some benchmarking purposes because of communication overhead or other agent-specific factors. For example, with Multi-ICE the request is passed back to the host for execution. This can lead to unpredictable delays in transmission and process scheduling.

Use this function to calculate time intervals (the length of time some action took) by calculating differences between intervals with and without the code sequence to be timed

Some systems allow more accurate timing (see *SYS_ELAPSED (0x30)* on page 5-26 and *SYS_TICKFREQ (0x31)* on page 5-26).

Entry

Register r1 must contain zero. There are no other parameters.

Return

On exit, r0 contains:

- the number of centiseconds since some arbitrary start point, if the call is successful
- –1 if the call is unsuccessful (for example, because of a communications error).

5.4.16 SYS_TIME (0x11)

Returns the number of seconds since 00:00 January 1, 1970. This is real-world time, regardless of any ARMulator configuration.

Entry

There are no parameters. Register r1 must contain zero.

Return

On exit, r0 contains the number of seconds.

5.4.17 SYS_SYSTEM (0x12)

Passes a command to the host command-line interpreter. This enables you to execute a system command such as `dir`, `ls`, or `pwd`. The terminal I/O is on the host, and is not visible to the target.

Caution

The command passed to the host is actually executed on the host. Ensure that any command passed will have no unintended consequences.

Entry

On entry, r1 contains a pointer to a two-word argument block:

word 1	This points to a string that is to be passed to the host command-line interpreter.
word 2	This is the length of the string.

Return

On exit, r0 contains the return status.

5.4.18 SYS_ERRNO (0x13)

Returns the value of the C library `errno` variable associated with the host implementation of the semihosting SWIs. The `errno` variable can be set by a number of C library semihosted functions, including:

- SYS_REMOVE
- SYS_OPEN
- SYS_CLOSE
- SYS_READ
- SYS_WRITE
- SYS_SEEK.

Whether `errno` is set or not, and to what value, is entirely host-specific, except where the ANSI C standard defines the behavior.

Entry

There are no parameters. Register r1 must be zero.

Return

On exit, r0 contains the value of the C library `errno` variable.

5.4.19 SYS_GET_CMDLINE (0x15)

Returns the command line used to call the executable.

Entry

On entry, r1 points to a two-word data block to be used for returning the command string and its length:

word 1	This is a pointer to a buffer of at least the size specified in word two.
word 2	This is the length of the buffer in bytes.

Return

On exit:

- Register r1 points to a two-word data block:

word 1	This is a pointer to null-terminated string of the command line.
word 2	This is the length of the string.

  The debug agent might impose limits on the maximum length of the string that can be transferred. However, the agent must be able to transfer a command line of at least 80 bytes.

  In the case of the Angel debug monitor using ADP, the minimum is slightly more than 200 characters.

- Register r0 contains an error code:
  - 0 if the call is successful
  - –1 if the call is unsuccessful (for example, because of a communications error).

5.4.20 SYS_HEAPINFO (0x16)

Returns the system heap parameters. The values returned are typically those used by the C library during initialization. For ARMulator, the values returned are the those provided in `bin\peripherals.ami`. For Multi-ICE, the values returned are the image location and the top of memory.

The C library can override these values (see *ADS Compiler, Linker, and Utilities Guide* for more information on memory management in the C library).

This call returns sensible answers, but the host debugger determines the actual values by using the `$top_of_memory` debugger variable.

Entry

On entry, r1 contains the address of a pointer to a four-word data block. Word 1 of the data block does not have to have a value. The contents of the data block are filled by the function. See Example 5-1 for the structure of the data block and return values.

Example 5-1

```
struct block {
    int heap_base;
    int heap_limit;
    int stack_base;
    int stack_limit;
    };
struct block *mem_block, info;
mem_block = &info;
AngelSWI(SYS_HEAPINFO, (unsigned) &mem_block);
```

Return

On exit, r1 contains the address of the pointer to the structure.

If one of the values in the structure is 0, the system was unable to calculate the real value. Typical values for an ARM development board are shown in Example 5-2 on page 5-25.

Example 5-2

```
Heap Base = 0x00000000
Heap Limit = 0x00076E00
Stack Base = 0x00078E00
Stack Limit = 0x00076E00
```

Copyright © 1999, 2000 ARM Limited. All rights reserved.

5.4.21 SYS_ELAPSED (0x30)

Returns the number of elapsed target ticks since the support code started execution. Ticks are defined by SYS_TICKFREQ. If the target cannot define the length of a tick, it can supply SYS_ELAPSED.

Entry

Register r1 contains a pointer to a double word for storing the number of elapsed ticks. The first word is the least significant word. The last word is the most significant word.

Return

If the double word pointed to by r1 (low-order word first) does not contain the number of elapsed ticks, r0 is set to –1.

5.4.22 SYS_TICKFREQ (0x31)

Returns the tick frequency.

Entry

Register r1 must contain 0 on entry to this routine.

Exit

On exit, r0 contains either:

- the number ticks per second
- –1 if the target does not know the value of one tick.

5.5 Debug agent interaction SWIs

In addition to the C library semihosted functions described in *Semihosting SWIs* on page 5-11, the following SWIs support interaction with the debug agent:

- The ReportException SWI. This SWI is used by the semihosting support code as a way to report an exception to the debugger.
- The EnterSVC SWI. This SWI sets the processor to Supervisor mode.
- The reason_LateStartup SWI. This SWI is obsolete and no longer supported.

These are described below.

5.5.1 angel_SWIreason_EnterSVC (0x17)

Sets the processor to Supervisor (SVC) mode and disables all interrupts by setting both interrupt mask bits in the new CPSR. With RealMonitor, Angel, or Multi-ICE, the User stack pointer (r13_USR) is copied to the Supervisor stack pointer (r13_SVC) and the I and F bits in the current CPSR are set, disabling normal and fast interrupts.

Note

If debugging with ARMulator:

- r0 is set to zero indicating that no function is available for returning to User mode
- the User mode stack pointer is *not* copied to the Supervisor stack pointer.

Entry

Register r1 is not used. The CPSR can specify User or Supervisor mode.

Return

On exit, r0 contains the address of a function to be called to return to User mode. The function has the following prototype:

```
void ReturnToUSR(void)
```

If EnterSVC is called in User mode, this routine returns the caller to User mode and restores the interrupt flags. Otherwise, the action of this routine is undefined.

If entered in User mode, the Supervisor stack is lost as a result of copying the user stack pointer. The return to User routine restores r13_SVC to the Angel Supervisor mode stack value, but this stack must not be used by applications.

After executing the SWI, the current link register will be r14_SVC, not r14_USR. If the value of r14_USR is required after the call, it must be pushed onto the stack before the call and popped afterwards, as for a `BL` function call.

5.5.2 angel_SWIreason_ReportException (0x18)

This SWI can be called by an application to report an exception to the debugger directly. The most common use is to report that execution has completed, using `ADP_Stopped_ApplicationExit`.

Entry

On entry r1 is set to one of the values listed in Table 5-3 and Table 5-4 on page 5-29. These values are defined in `adp.h`.

The hardware exceptions are generated if the debugger variable `$vector_catch` is set to catch that exception type, and the debug agent is capable of reporting that exception type. Angel cannot report exceptions for interrupts on the vector it uses itself.

Table 5-3 Hardware vector reason codes

Name (#defined in adp.h)	Hexadecimal value
`ADP_Stopped_BranchThroughZero`	`0x20000`
`ADP_Stopped_UndefinedInstr`	`0x20001`
`ADP_Stopped_SoftwareInterrupt`	`0x20002`
`ADP_Stopped_PrefetchAbort`	`0x20003`
`ADP_Stopped_DataAbort`	`0x20004`
`ADP_Stopped_AddressException`	`0x20005`
`ADP_Stopped_IRQ`	`0x20006`
`ADP_Stopped_FIQ`	`0x20007`

Exception handlers can use these SWIs at the end of handler chains as the default action, to indicate that the exception has not been handled.

Table 5-4 Software reason codes

Name (#defined in adp.h)	Hexadecimal value
`ADP_Stopped_BreakPoint`	`0x20020`
`ADP_Stopped_WatchPoint`	`0x20021`
`ADP_Stopped_StepComplete`	`0x20022`

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Table 5-4 Software reason codes (continued)

Name (#defined in adp.h)	Hexadecimal value
`ADP_Stopped_RunTimeErrorUnknown`	*0x20023
`ADP_Stopped_InternalError`	*0x20024
`ADP_Stopped_UserInterruption`	0x20025
`ADP_Stopped_ApplicationExit`	0x20026
`ADP_Stopped_StackOverflow`	*0x20027
`ADP_Stopped_DivisionByZero`	*0x20028
`ADP_Stopped_OSSpecific`	*0x20029

* next to values in Table 5-4 indicates that the value is not supported by the ARM debuggers. The debugger reports an `Unhandled ADP_Stopped exception` for these values.

Return

No return is expected from these calls. However, it is possible for the debugger to request that the application continue by performing an RDI_Execute request or equivalent. In this case, execution continues with the registers as they were on entry to the SWI, or as subsequently modified by the debugger.

5.5.3 angel_SWIreason_LateStartup (0x20)

This SWI is obsolete.

Glossary

The items in this glossary are listed in alphabetical order, with any symbols and numerics appearing at the end.

ADP	See *Angel Debug Protocol*.
ADS	See *ARM Developer Suite*.
ADU	See *ARM Debugger for UNIX*.
Advanced Microcontroller Bus Architecture	On-chip communications standard for high-performance 32-bit and 16-bit embedded microcontrollers.
ADW	See *ARM Debugger for Windows*.
AMBA	See *Advanced Microcontroller Bus Architecture*.
Angel	Angel is a program that enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.
Angel Debug Protocol	Angel uses a debugging protocol called the Angel Debug Protocol (ADP) to communicate between the host system and the target system. ADP supports multiple channels and provides an error-correcting communications protocol.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

ARM Debugger for UNIX	ARM Debugger for UNIX (ADU) and ARM Debugger for Windows (ADW) are two versions of the same ARM debugger software, running under UNIX or Windows respectively. This debugger was issued originally as part of the ARM Software Development Toolkit. It is still fully supported and is now supplied as part of the ARM Developer Suite.
ARM Debugger for Windows	ARM Debugger for Windows (ADW) and ARM Debugger for UNIX (ADU) are two versions of the same ARM debugger software, running under Windows or UNIX respectively. This debugger was issued originally as part of the ARM Software Development Toolkit. It is still fully supported and is now supplied as part of the ARM Developer Suite.
ARM Developer Suite	A suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.
ARM eXtended Debugger	The ARM eXtended Debugger (AXD) is the latest debugger software from ARM that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. AXD is supplied in both Windows and UNIX versions.
ARM Symbolic Debugger	An interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language. It is a command-line debugger that runs on all supported platforms.
armsd	*See* ARM Symbolic Debugger.
ARMulator	ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
AXD	See *ARM eXtended Debugger*.
Big-endian	Memory organization where the least significant byte of a word is at a higher address than the most significant byte. See also *Little-endian*.
Breakpoint	A location in the image. If execution reaches this location, the debugger halts execution of the image. See also *Watchpoint*.
Context	The information stored in a block of registers on entry to a subroutine, and held there until required for restoring the information on exit from the subroutine.
Coprocessor	An additional processor which is used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.
CPSR	Current Program Status Register. See *Program Status Register*.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Debugger	An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.
DLL	See *Dynamic Linked Library*.
Double word	A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
Dynamic Linked Library	A collection of programs, any of which can be called when required by an executing program. A small program that helps a larger program communicate with a device such as a printer or keyboard is often packaged as a DLL.
ELF	*See* Executable and linking format.
Executable image	*See* Image.
Executable and linking format	The industry standard binary file format used by the ARM Developer Suite. ELF object format is produced by the ARM object producing tools such as armcc and armasm. The ARM linker accepts ELF object files and can output either an ELF executable file, or partially linked ELF object.
Function	A C++ method or free function.
Halfword	A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
Host	A computer which provides data and other services to another computer.
ICE	In-Circuit Emulator.
Image	A file of executable code which can be loaded into memory on a target and executed by a processor there.
Integrator™	The ARM Integrator development board.
Joint Test Access Group	Many debug and programming tools use a JTAG interface port to communicate with processors. For further information refer to IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).
JTAG	*See* Joint Test Access Group.
Little-endian	Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also *Big-endian*.
Memory management unit	Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.

MMU	See *Memory Management Unit*.
Multi-ICE	Multi-processor in-circuit emulator. ARM registered trademark.
Processor	An actual processor, real or emulated running on the target. A processor always has at least one context of execution.
Processor Status Register	*See* Program Status Register.
Profiling	Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code. *Call-graph profiling* provides great detail but slows execution significantly. *Flat profiling* provides simpler statistics with less impact on execution speed. For both types of profiling you can specify the time interval between statistics-collecting operations.
Program Status Register	*Program Status Register* (PSR), containing some information about the current program and some information about the current processor. Often, therefore, also referred to as *Processor Status Register*. Is also referred to as *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR). The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned.
Program image	*See* Image.
Protection Unit	Hardware that controls caches and access permissions to blocks of memory.
PSR	*See* Program Status Register.
PU	*See* Protection Unit.
RDI	*See* Remote Debug Interface.
Remote_A	A communications protocol used, for example, between debugger software such as ARM eXtended Debugger (AXD) and a debug agent such as Angel.
Remote Debug Interface	An open ARM standard procedural interface between a debugger and the debug agent. The widest possible adoption of this standard is encouraged.
Saved Program Status Register	*See* Program Status Register

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Semihosting	A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.
Software Interrupt	SWI. An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.
Source File	A file which is processed as part of the image building process. Source files are associated with images.
SPSR	Saved Program Status Register. See *Program Status Register*.
SWI	*See* Software Interrupt.
Target	The target processor (real or simulated), on which the target application is running.
	The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.
Task Queue Item	Angel context switching information.
TQI	See *Task Queue Item.*
Tracing	Recording diagnostic messages in a log file, to show the frequency and order of execution of parts of the image. The text strings recorded are those that you specify when defining a breakpoint or watchpoint. See *Breakpoint* and *Watchpoint*. See also *Stack backtracing*.
Veneer	A small block of code used with subroutine calls when there is a requirement to change processor state (ARM to Thumb or Thumb to ARM) or branch to an address that cannot be reached in the current processor state.
Veneer memory model	A memory model that adds its own functionality to another memory model. It calls the other memory model for part of its functionality.
Watchpoint	A location in the image that is monitored. If the value stored there changes, the debugger halts execution of the image. See also *Breakpoint*.
Word	A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Copyright © 1999, 2000 ARM Limited. All rights reserved.