ARM Developer Suite

Version 1.0.1

Debug Target Guide

ARM

ARM DUI0058B

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

Proprietary Notice

Contents
Debug Target Guide

Preface

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

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.

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 *Angel*

Read this chapter for a description of Angel, the ARM debug monitor.

Chapter 6 *Semihosting SWIs*

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 may be entered at the keyboard, such as commands, file and program names, and source code.

`type`writer Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.

`typewriter italic`
Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.

`typewriter bold`
Denotes language keywords when used outside example code and ARM processor signal names.

Further reading

This section lists publications from 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:

- *Getting Started* (ARM DUI 0064A)
- *ADS Tools Guide* (ARM DUI 0067A)
- *ADS Debuggers Guide* (ARM DUI 0066A)
- *ADS Developer Guide* (ARM DUI 0056A).

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

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

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

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

Feedback

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

Feedback on the ARM Developer Suite

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

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

Feedback on this book

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

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

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the debug support facilities provided in the ADS version 1.00. It contains the following sections:

1.1 About debug support

The debug support component of ADS consists of ARMulator and Angel.

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 may be either:

- a software simulator, simulating ARM hardware
- real ARM-based hardware.

ARMulator is a simulator that runs on the same host computer as the debugger (see *ARMulator* on page 1-3).

ARM-based hardware could be an ARM evaluation or development board, a third-party board, or ARM-based hardware of your own design. In addition to the software that you are developing, it may need to run a *debug monitor* to communicate with the debugger.

Angel is the debug monitor supplied with ADS (see *Angel* on page 1-3).

ARMulator and the debug monitor use *software interrupts* (SWI) and the host computer to handle process requests from the application for initialization, memory management, and I/O. Using the host computer to assist the local function calls is called *semihosting* (see *Semihosting SWIs* on page 1-3).

In-circuit emulators such as EmbeddedICE and Multi-ICE use an alternative method instead of Angel. See the documentation accompanying the hardware for details.

1.2 ARMulator

ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.

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

1.3 Angel

Angel is a debug monitor. It is designed to help you to develop and debug applications running on ARM-based hardware. Using Angel you can debug applications running in either ARM state or Thumb state.

You can use Angel to:

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

See the following chapters for more information:

- Chapter 5 *Angel*
- Chapter 6 *Semihosting SWIs*.

1.4 Semihosting SWIs

The semihosting SWIs provide the mechanism for using applications in a semihosted environment. SWI handling is available for both ARM and Thumb.

You can use the semihosting SWIs to produce applications which work with Angel, ARMulator, or your own SWI handler.

See Chapter 6 *Semihosting SWIs* for more information.

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 is a collection of modules that simulate 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 Tools Guide* for more information on C and C++ library support. See also Chapter 6 *Semihosting SWIs* for information on the C library semihosting SWIs supported by ARMulator.

ARMulator does not support ARM10 in the same way that it supports other processors. Refer to *Basic ARM ten system* on page 2-31 for further information.

2.2 ARMulator components

This section does not apply to ARM10 systems. For information about ARM10 systems, see *Basic ARM ten system* on page 2-31.

ARMulator consists of a series of modules. The main ones 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, without rebuilding ARMulator (see *Configuring ARMulator* on page 2-4 and *Memory model with memory map* on page 2-20).

You can write your own modules, or edit copies of the predefined ones, if the modules provided do not meet your needs. 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 most of the modules, excluding the processor models, is supplied. You can use these as examples to help you write your own modules (see Chapter 3 *Writing ARMulator Models*).

2.2.1 Configuring ARMulator

You can configure some of the details of ARMulator from `armsd`, AXD, ADU, or ADW. See *ADS Debuggers Guide* for details. The currently active models and configurations are announced in the debugger startup banner

To make other adjustments you must edit ARMulator files directly. These are described in this chapter, in Chapter 3 *Writing ARMulator Models*, and Chapter 4 *ARMulator Reference*.

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

2.3 Tracer

You can use Tracer to trace instructions, memory accesses, and events. The configuration file `armul.cnf` controls what is traced (see *armul.cnf, the ARMulator configuration file* on page 4-98).

Note

Tracer cannot be used with ARM10 systems. See *Basic ARM ten system* on page 2-31 for information about ARM10 systems.

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

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

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 may 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 `armul.cnf`.

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 4-91.

2.3.3 Configuring Tracer

Tracer has its own section in the ARMulator configuration file (`armul.cnf`):

```
{ 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
TraceMemory=False
TraceIdle=False
TraceNonAccounted=False
TraceEvents=False
;; Where to trace memory - if not set, it will trace at the
;; core.
TraceBus=True
;; Flags - disassemble instructions; start 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:

`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.

`TraceBus` controls the trace data source. This is one of:

- `TRUE` Bus (between processor and memory)
- `FALSE` Core (between core and cache, if present).

For more information, see *ARMul_InstallMemoryInterface* on page 4-8.

`Disassemble` disassembles instructions. Simulation is much slower if you enable disassembly.

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 4-91 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.

Note

Profiler cannot be used with ARM10 systems. See *Basic ARM ten system* on page 2-31 for information about ARM10 systems.

2.4.1 Configuring Profiler

The `Profiler` section in the configuration file is as follows:

```
{ Profiler
;; For example - to profile the PC value when cache misses
;; happen, set:
;Type=Event
;Event=0x00010001
;EventWord=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 4-91 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 4-91.

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

2.5 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 `UsePageTables` tag in the configuration file, `armul.cnf`, and set it to `True` or `False` as appropriate:

```
UsePageTables=True
```

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

```
{ 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.

2.5.1 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. `FastBus` is set to No by default. You can set it to Yes using the pagetables section of `armul.cnf`, or a write to CP15.

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

2.5.2 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.

2.5.3 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, `armul.cnf` 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.

`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.

2.5.4 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 `armul.cnf`.

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.

2.5.5 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.

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.

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.

2.6 Flat memory model

ARMflat 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 flat memory model does not generate aborts.

ARMflat is the default memory model used if you do not:

- specify a mapfile or validation model in AXD, ADU, or ADW
- edit `armul.cnf`.

2.6.1 Selecting the flat memory model

You select the flat model by setting `Default=Flat` in the `Memories` section of the `armul.cnf` file:

```
{ Memories

;; Default memory model is the "Flat" model, or the "MapFile"
;; model if there is an armsd.map file to load.

; Validation suite uses the trickbox
#if Validate
Default=TrickBox
#endif

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

;; Default default is the flat memory map
Default=Flat
```

2.7 Fast memory model

ARMfast is a flat memory model of 2MB of RAM. Simulation using ARMfast is typically 17% faster than for ARMflat. This performance increase is partly achieved by not counting cycles, so cycle counts shown by `$statistics` in the debugger will be zero. This model is intended for use by software developers who want maximum simulation speed, and are not interested in counting cycle or measuring execution time.

The memory size is limited to 2MB. You can change this by editing `armfast.c` and rebuilding ARMulator, as described in *Rebuilding ARMulator* on page 3-11.

The fast memory model does not generate aborts.

2.7.1 Selecting the fast memory model

You select the fast memory model by setting `Default=Fast`, in the `Memories` section of the `armul.cnf` file:

```
{ Memories

;; Default memory model is the "Flat" model, or the "MapFile"
;; model if there is an armsd.map file to load.

;; Validation suite uses the trickbox
#if Validate
Default=TrickBox
#endif

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

;; Default default is the flat memory map
;Default=Flat
Default=Fast
```

2.8 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 4-94).

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 may generate aborts if you specify a memory region with access type as - (hyphen).

2.8.1 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.

2.8.2 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
```


2.8.3 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 should specify 80ns for 70ns RAM if there is a 10ns propagation delay.

2.8.4 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 `armul.cnf` file:

```
{ MapFile
;; Options for the mapfile memory model
CountWaitStates=True
AMBABusCounts=False
SpotISCycles=True
ISTiming=Early
}
```

Counting wait states

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

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 `armul.cnf`.

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 `armul.cnf`. 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.

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 `armul.cnf`.

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.

2.9 DummyMMU

DummyMMU is a dummy implementation of an ARM architecture v3 or v4 coprocessor 15. It does not provide any of the cache and MMU functions, but does prevent accesses to it being Undefined Instruction exceptions.

Reads from register 0 return a dummy identification register value. You can configure the value to be returned.

Writes to register 1 of the dummy coprocessor (the system configuration register) set the **bigend**, **lateabt** and other signals.

2.9.1 Configuring DummyMMU

You can set the code of DummyMMU in the configuration file. Use the following entry in the `Coprocessors` section of `armul.cnf`:

```
{ Coprocessors

; Here is the list of coprocessors, in the form
;; Coprocessor[<n>]=Name

#if COPROCESSOR_DummyMMU
;; By default, install a dummy MMU on coprocessor 15.
CoProcessor[15]=DummyMMU

; Here is the configuration for the coprocessors.
;; DummyMMU can be configured to return a given Chip ID
;DummyMMU:ChipID=
#endif
}
```

The line:

```
;DummyMMU:ChipID=
```

can be uncommented and set to any value. For example, to configure DummyMMU to return the ARM710 ID code (`0x41017100`), change this line to:

```
; Here is the configuration for the coprocessors.
;; DummyMMU can be configured to return a given Chip ID
DummyMMU:ChipID=0x41017100
```

2.10 Angel

The Angel Debug Monitor is a program which runs on ARM-based hardware. It handles communication between your prototype software, running on the same ARM-based hardware, and a debugger running on your host machine.

When you develop your prototype software on the ARMulator, you can use the Angel operating system model to simulate the Angel Debug Monitor.

2.10.1 Configuring Angel

The configuration for the Angel model is in the `armul.cnf` file. Look for:

```
{ OS
;; Angel configuration
[ ...]
}
```

The configuration options are:

```
AngelSWIARM=0x123456
AngelSWIThumb=0xab
HeapBase=0x40000000
HeapLimit=0x70000000
StackBase=0x80000000
StackLimit=0x70000000
```

where:

`AngelSWIARM`/`AngelSWIThumb`
: declares the SWI numbers that Angel uses. For descriptions, see Chapter 6 *Semihosting SWIs*.

`HeapBase`/`HeapLimit`
: defines the application heap.

`StackBase`/`StackLimit`
: defines the application stack.

The following options define the initial locations of the exception mode stack pointers.

```
AddrSuperStack=0xa00
AddrAbortStack=0x800
AddrUndefStack=0x700
AddrIRQStack=0x500
AddrFIQStack=0x400
```


The semi-hosting C library changes the stack pointer to the value returned by `SWI_SYSHEAPINFO`. `SWI_SYSHEAPINFO` is set to the value of `StackBase` configured above. You can specify the location of the User mode stack by editing the address in `AddrUserStack`:

```
AddrUserStack=0x80000
```

These options define the location in memory where ARMulator places the code to handle the hardware exception vectors:

```
AddrSoftVectors=0xa40
AddrsOfHandlers=0xad0
SoftVectorCode=0xb80
```

The final option points to a buffer where the Angel model places a copy of the command line. This can be retrieved by catching the `RDI_Info` call, `RDISet_Cmdline`:

```
AddrCmdLine=0xf00
```

Note

The default heap/stack model used by the C library ignores `HeapLimit` and `StackLimit`. See the libraries chapter in *ADS Tools Guide* for details.

2.11 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*.

Note

This section does not apply to ARM10 systems. See *Basic ARM ten system* on page 2-31 for information about peripheral models in ARM10 systems.

2.11.1 Configuring ARMulator to use the peripheral models

Enable or disable each peripheral model by changing the relevant entry in the `armul.cnf` file:

```
;; ************************************************************
;; ARMulator Peripheral Models
;; Central list of peripherals
;; Use this list to enable/disable peripherals
;; ************************************************************
;; To enable a peripheral change the rhs to TRUE
;; To disable a peripheral change the rhs to FALSE
TimerEnabled=False
WDogEnabled=False
IntCEnabled=False
```

2.11.2 Switch

The switch is a model of an address decoder or memory or peripheral controller. It is a configurable address decoder that makes it easier to attach peripheral models without drastically reducing the performance of ARMulator.

The switch is a veneer between the processor core and memory. It routes memory accesses to the appropriate memory model. Routing is based on the access address and a set of memory address ranges, peripheral address ranges, and peripheral address masks.

The switch is installed if any of the reference peripheral models is enabled.

2.11.3 Interrupt controller

In addition to `IntCEnabled`, the interrupt controller has the following configuration items:

```
{ InterruptController
Range=0x0a000000,0x0a00010c
;; set WARN to enable warnings about invalid register accesses
WARN=FALSE
WAITS=1
}
```

`Range` 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 4-121.

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

2.11.4 Timer

In addition to `TimerEnabled`, the timer has the following configuration items:

```
{ TimerCounter
Range=0x0a800000,0x0a80003f
;; The RPS Clock. This is usually the processor clock rate
CLK=20000000
;; Interrupt controller source bits - 4 and 5 as standard
IntOne=4
IntTwo=5
;; set WARN to enable warnings about invalid register accesses
WARN=FALSE
WAITS=1
}
```

`Range` specifies the area in memory into which the timer registers are mapped. For details of the interrupt controller registers, see *Timer* on page 4-123.

`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.

2.11.5 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.

In addition to `WDogEnabled`, Watchdog has the following configuration items:

```
{ WatchDog
Range=0xb0000000,0xb0000004
KeyValue=0x12345678
WatchPeriod=0x80000
IRQPeriod=3000
IntNumber=16
StartOnReset=True
RunAfterBark=True
;; set WARN to enable warnings about invalid register accesses
WARN=FALSE
WAITS=1
}
```

`Range` 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.

2.12 Other models

This section gives basic user information about the less complex models. For more detailed information, refer to Chapter 4 *ARMulator Reference*.

Note

This section does not apply to ARM10 systems. See *Basic ARM ten system* on page 2-31 for information about peripheral models in ARM10 systems.

2.12.1 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.

To enable the stack tracker, edit `armul.cnf`.

1. Find the line:

   ```
   TrackStack=False
   ```

2. Change it to:

   ```
   TrackStack=True
   ```

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

```
;; The StackUse model continually monitors the stack pointer and
;; reports the amount of stack used in $statistics. It needs to
;; be configured with the stack's location.
{ 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 Angel model (see *Configuring Angel* on page 2-24).

2.12.2 Windows Hourglass

This model calls the debugger regularly during execution. This is required when you are using AXD, ADU or ADW. If you want to alter the interval between calls to the debugger, find the section listed below in `armul.cnf` and edit it.

```
{ WindowsHourglass
;; We can control how regularly we callback the frontend
;; More often (lower value) means a slower simulator, but
;; faster response. The default is 8192.
Rate=8192
}
```

2.12.3 Watchpoints

Watchpoints provides memory watchpoints. It is a veneer between the processor core and memory, or cache as appropriate.

To enable watchpoints, change the `WatchpointsEnabled` line in `armul.cnf`:

```
;; To enable watchpoints, set "WatchPointsEnabled"
WatchpointsEnabled=True
```

Disable watchpoints when benchmarking code. Enable watchpoints when debugging code using watchpoints.

2.12.4 Validate

This is a small coprocessor that is used to validate the behavior of the ARM simulator. It can cause interrupts and busy-waits, for example.

2.12.5 Trickbox

This is a memory model of a system where accessing various addresses causes events, such as aborts and interrupts.

2.12.6 Bytelane

This is a veneer memory model. It is a veneer between the processor and the real memory model. It converts accesses from the core into byte-lane accesses. Byte-lane accesses are also known as byte-strobe accesses.

2.12.7 ARM PIE

This is a model of the ARM PIE card. It is only available for UNIX.

2.13 Basic ARM ten system

Basic ARM Ten System (BATS) is a separate modelling scheme from ARMulator. You cannot use any of the other ARMulator models, such as Profiler or Tracer, with BATS.

To use BATS:

1. Select BATS instead of ARMulator as your debugger target.
2. Select a *configuration trace file* (CTR file) from your debugger interface.
3. Either:
   - specify the name of your chosen CTR file when you start `armsd`
   - select the name of a CTR file from the list in the BATS configuration window in AXD, ADU, or ADW.

See *ADS Debuggers Guide* for details.

2.13.1 Configuration trace files

CTR files describe the configurations of the systems that BATS can model. They describe which components are used by the system and how they are interconnected. Some components have configuration options which are specified in the CTR file.

Three CTR files are supplied with BATS. See:

- *ARM1020T* on page 2-32
- *ARM1020T_PERIP* on page 2-33
- *ARMv5TM* on page 2-35.

You can also write your own CTR files. You can do this by copying one of the supplied files and editing the copy. See *Basic ARM ten system configuration trace files* on page 4-114 for more information.

In all the supplied configurations:

- the heap occupies `0x30000000` to `0x70000000` of the address space
- the stack occupies `0x80000000` to `0x70000000` of the address space
- the time resolution is 10ps.

2.13.2 ARM1020T

The ARM1020T configuration trace file, ARM1020T.ctr, defines a model of an ARM1020T system with two memory modules. It has:

- a system coprocessor (cp15)
- a memory management unit
- a write buffer
- separate instruction and data caches.

Communication between the caches or write buffer and the external memory modules is through the AMBA bus. This system is shown in Figure 2-1. The filenames of BATS modules are shown in brackets. For further information see *Creating instances* on page 4-115.

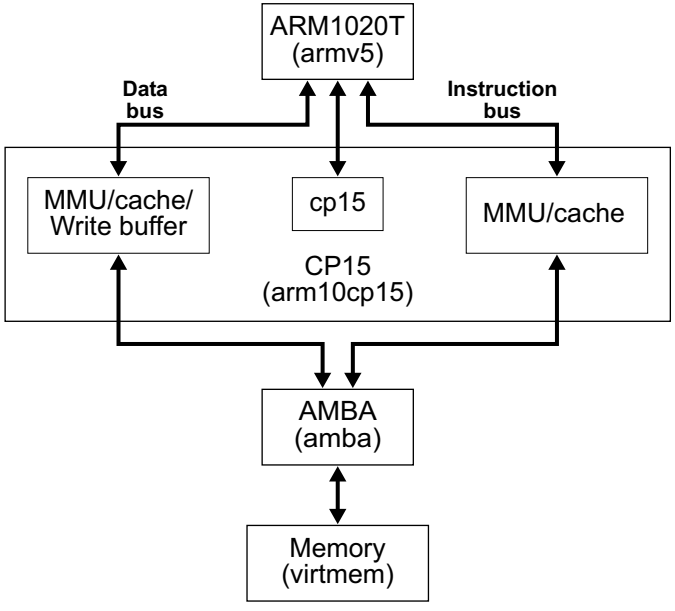

Figure 2-1 ARM1020T system

2.13.3 ARM1020T_PERIP

The ARM1020T_PERIP configuration trace file, `ARM1020T_PERIP.ctr`, defines a model of an ARM1020T system with two memory modules and a set of reference peripherals. It has:

- a system coprocessor (cp15)
- a memory management unit
- a write buffer
- separate instruction and data caches.

Communication between the caches or write buffer and the external memory modules is through the AMBA bus. This system is shown in Figure 2-2. The filenames of BATS modules are shown in brackets. For further information see *Creating instances* on page 4-115.

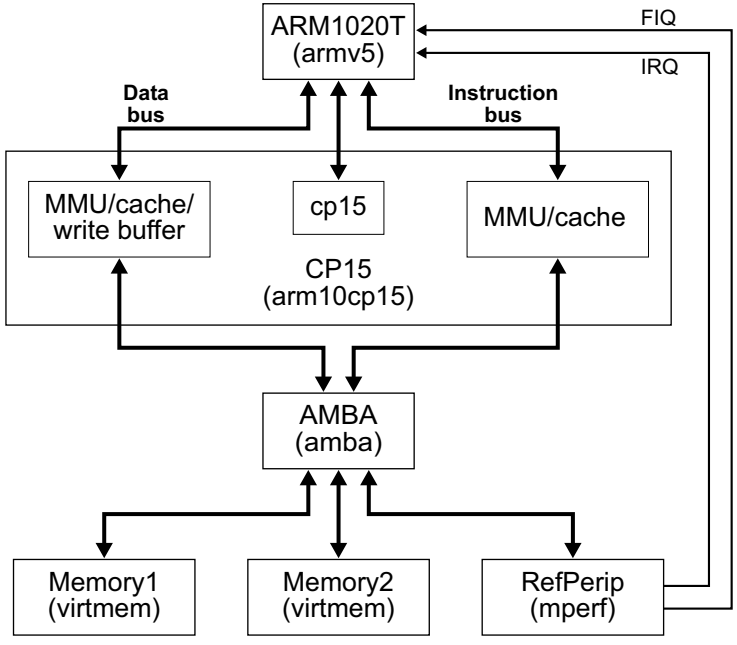

Figure 2-2 ARM1020T_PERIP system

Reference peripherals

The following facilities of the reference peripherals are implemented:

- *Interrupt controller* on page 4-121.
- *Timer* on page 4-123.

See *Reference Peripherals Specification* for additional details.

ARM1020T_PERIP memory map

In ARM1020T_PERIP, as shown in Figure 2-3:

- Memory1 occupies `0x00000000` to `0x07ffffff` of the address space
- the reference peripherals occupy `0x0a000000` to `0x0dffffff` of the address space
- Memory2 occupies `0x10000000` to `0x7fffffff` of the address space.

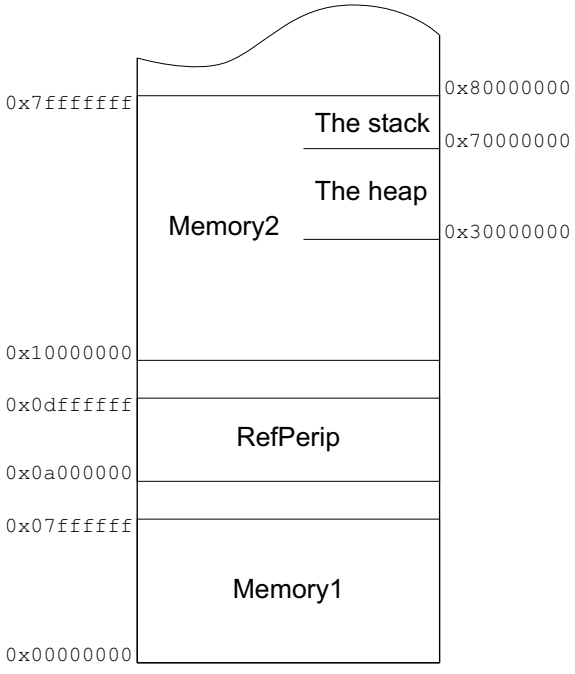

Figure 2-3 Memory map of ARM1020T_PERIP

2.13.4 ARMv5TM

The ARMv5TM configuration trace file, ARMv5TM.ctr, defines a model of a generic ARM architecture v5 system with one memory module. It has no coprocessor 15 or caches.

Note

This model does not correspond to any real hardware. It cannot be used for benchmarking. It is supplied for software development and debugging purposes only.

Although hardware without caches would be slower if it existed, the simulation is faster without the caches.

This system is shown in Figure 2-4. The filenames of BATS modules are shown in brackets. For further information see *Creating instances* on page 4-115.

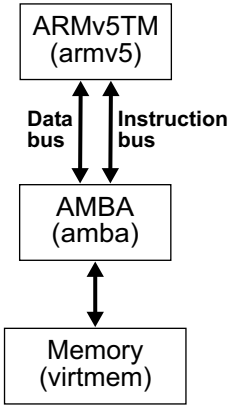

Figure 2-4 ARMv5TM system

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 Adding models to ARMulator

This chapter does not contain information about the *Basic ARM Ten System* (BATS). For information about BATS, see *Basic ARM ten system* on page 2-31.

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 can be found in the rebuild kit on UNIX in:

Install_Directory`\ARMulate`

or on PC in:

Install_Directory`\ARMulate\Win32\ARMulate\user`

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 3-4).

You can make a copy of one of these models and edit the copy.

To use your new model, you must rebuild ARMulator using the appropriate compiler (see *Rebuilding ARMulator* on page 3-11).

3.1.1 Model stubs

Basic models, memory models, coprocessor models, and operating system models attach to ARMulator through a stub. A stub consists of an initialization function and a textual name for the model. ARMulator uses the textual name to locate the initialization function.

Basic models can be initialized either before or after memory models are initialized. This means that there are two distinct types of basic model:

- early models
- late models.

ARMulator locates each model in turn, calling the initialization function of each model, and passing in a pointer to a structure containing a list of pointers.

Each model fills in this set of function pointers at initialization time. The model should also register an `ExitUpcall()` (see *ExitUpcall* on page 4-55) during initialization, to free any memory that it allocates.

Model initialization sequence

The model initialization functions are called in the following order:

1. ARMulator core model
2. Early basic models, such as peripheral models (see *Basic model interface* on page 4-4)
3. Memory models, including veneer memory models installed by an early basic model (see *Basic model interface* on page 4-4)
4. Coprocessor models
5. Operating system models
6. Late basic models (see *Basic model interface* on page 4-4).

3.1.2 Supplied models

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

- *Basic models*
- *Peripheral models*
- *Memory models* on page 3-5
- *Coprocessor models* on page 3-5
- *Operating system models* on page 3-5.

Basic models

`tracer.c` The tracer module can trace instruction execution and events from within ARMulator (see *Tracer* on page 4-89). You can link your own tracing code onto the tracer module, allowing real-time tracing for example. tracer is an early basic model.

`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 *UnkRDIInfoUpcall* on page 4-62). `profiler` is a late basic model.

`pagetab.c` On reset, this module sets up cache, PU or MMU and associated pagetables inside ARMulator (see *Pagetable module* on page 2-12). `pagetab` is a late basic model.

`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 `armul.cnf` file. `stackuse` is a late basic model.

Peripheral models

All the peripheral models are early basic models.

`intc.c` See *Interrupt controller* on page 2-27. `intc` is a model of the interrupt controller peripheral described in the *Reference Peripherals Specification* (RPS).

`timer.c` See *Timer* on page 2-27. `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`.

`watchdog.c` Watchdog. See *Watchdog* on page 2-28. `watchdog` is a generic watchdog model. It does not model any specific watchdog hardware, but provides generic watchdog functions.

Memory models

The following source files are provided for memory models:

`armflat.c` This module implements a flat model of 4GB RAM.

`armfast.c` This module implements a flat model of 2MB RAM.

`armmap.c` This memory model allows you to specify memory layout using an `armsd.map` file, see *Map files* on page 4-94. This slows down simulation speed, so when no `armsd.map` file is present, ARMulator uses the faster `armflat.c` model in preference.

`excache.c` `excache` is an example of a basic cache model. See *Writing a new cache model* on page 3-7. `excache` provides a starting point for you to write your own cache models.

`tracer.c` As well as being a basic model, the tracer module provides a veneer memory model that can log memory accesses.

Coprocessor models

`dummymmu.c` This is a cut-down model of coprocessor 15 (the system coprocessor).

Operating system models

`angel.c` This is an implementation of the *Software Interrupts* (SWIs) and environment required for running programs linked with the semihosted C library on ARMulator.

`noos.c` This is a dummy operating system model, where no SWIs are intercepted.

3.2 Writing a new peripheral model

A peripheral model is an early model that is accessed via the switch memory model. The switch model carries out partial address decoding to select a memory device. An address range or an address mask is specified in the `armul.cnf` file for the switch to control which peripherals are accessed for an address.

A template file, called `template.c`, is provided in the `user` directory of the rebuild kit. There is a companion file called `notes.txt`. `notes.txt` is a step by step guide that explains how to write a peripheral model based on the supplied template.

3.3 Writing a new cache model

A cache memory model is a veneer between the processor model and the main memory model. Other veneer memory models, for which the source code is supplied, can be used as examples to help you write your own model. One example is `watchpnt.c`. Both the MMUlator and the StrongMMU cache simulators are part of the core ARMulator, and are not supplied in source form.

Memory models have two main parts:

- *Initialization* on page 3-8
- *Memory access* on page 3-10.

An example cache model, `excache.c`, is supplied in the ARMulator rebuild kit.

The file `excache.c` defines an extra memory model. For ARMulator to know about this model, you must declare the model in `models.h` by adding the line:

```
MEMORY(ExampleCache)
```

The reference `ExampleMemory` comes from `ARMul_MemStub ExampleCache` in the file `example.c`.

You must also add the object file to the supplied Makefile, along with a rule for building the model.

3.3.1 Initialization

A cache model must include the standard initialization functions, such as allocating a state, setting up the interface and so on. It must also:

- Use `ToolConf_Lookup(config, ARMulCnf_Memory)` to find the name of the memory model.
- Use `ARMul_FindMemoryInterface` to locate the memory model and initialize it. To do this, the cache model must have its own `ARMul_MemInterface` block.

Example 3-1 shows this.

Example 3-1 Cache model initialization

```
/* Find the name of the child memory interface */

child_name = (tag_t)ToolConf_Lookup(config, ARMulCnf_Memory);

/* Now locate it using ARMul_FindMemoryInterface. This also locates
 * its configuration for us
 */

if (child_name != NULL)
    child_init = ARMul_FindMemoryInterface(state, child_name, &child_config);
if (child_name == NULL || child_init == NULL || child_init == MemInit)
    return ARMul_RaiseError(state, ARMulErr_NoMemoryChild, ModelName);

/* Initialize the child model */

child_interf = &cache->child;
err = child_init(state, child_interf, type, child_config);
if (err != ARMulErr_NoError) {
    free(cache);
    return err;
}
```

Other memory interface functions such as `ReadClock` and `ReadCycles`, must be forwarded to the external memory model. You cannot just copy the functions over to `ARMul_MemInterface`, as they would have the wrong `handle`. You must create thin veneer functions.

Finding clock speed

To find the clock speed in a memory model, use code similar to the following:


```
const char *option;
unsigned long clockspeed;
bool hasClock;

option = ToolConf_Lookup(config, ARMulCnf_MCLK);
if (option == NULL) {
    hasClock = FALSE;
} else {
    hasClock = TRUE;
    clockspeed = ToolConf_Power(option, FALSE);
}
```

If you need to know the CPU's clock speed, rather than the memory clock speed, replace:

```
option = ToolConf_Lookup(config, ARMulCnf_MCLK);
```

with:

```
option = ToolConf_Lookup(config, Dbg_Cnf_CPUSpeed);
```

3.3.2 Memory access

In a cache model, the memory access function has to:

- Search the cache for the data being accessed.
- For a read:
  - if found, read from the cache, then perform an idle cycle on the external bus
  - if not found, perform a cache line fill.
- For a write:
  - if found, write the value to the cache, then perform the write on the external bus
  - if not found, perform the write on the external bus.

The details might vary if you wanted to model, for example, a write-back cache, a write buffer, or memory protection.

3.4 Rebuilding ARMulator

Which UNIX version of ARMulator you build depends on whether you are using Solaris or HP, and whether you want to use your rebuilt ARMulator with armsd or with ADU. In order to rebuild ARMulator you need the appropriate build tools:

PC You require Microsoft Visual C++ Version 5.0 to rebuild `armulate.dll`.

Solaris/armsd Build the armsd ARMulator executable using the GNU gcc compiler. The ADS1.0 version was built using gcc version egcs-2.91.66.

Solaris/ADU You require the SparcWorks C compiler version 4.2 to rebuild `ARMulate.dll`.

HP/armsd Build the armsd ARMulator executable using an ANSI-compliant C compiler.

HP/ADU You require the HP ANSI C compiler version 1.18 or later to rebuild `ARMulate.dll`.

You should make a copy of the ARMulate directory with a different name, and work in that.

3.4.1 Rebuilding on UNIX

In this section `example.c` is used to illustrate how to add a source file. Replace this with a filename of your own.

The locations of the source files and makefiles for the different UNIX versions of ARMulator are shown in Table 3-1. All the locations are relative to *`Install_Directory`*.

Table 3-1

	Source file location	Makefile
Solaris/armsd	`solaris/armsd/source`	`solaris/armsd/build/makefile`
Solaris/ADU	`solaris/source/ARMulate`	`solaris/source/ARMulate/makefile.sol2`
HP/armsd	`HP/armsd/source`	`HP/armsd/build/makefile`
HP/ADU	`HP/source/ARMulate`	`HP/source/ARMulate/makefile.hp700mt`

Follow these steps to add a file and rebuild ARMulator under UNIX:

1. Place the new source code file (or files) in the source directory.
2. Load the Makefile into an editor.
3. Find the entry:

   ```
   OBJALL=main.o angel.o armfast.o armflat.o armmap.o \
   armpie.o bytelane.o dummymmu.o ebsa110.o errors.o \
   models.o pagetab.o profiler.o tracer.o trickbox.o \
   validate.o watchpnt.o winglass.o switch.o intc.o timer.o \
   watchdog.o dcc.o tube.o stackuse.o cnffile.o
   ```

   (`OBJS` instead of `OBJALL` in the ADU versions).
4. Add the new object filename to the list of objects to link:

   ```
   OBJALL=main.o angel.o armfast.o armflat.o armmap.o \
   armpie.o bytelane.o dummymmu.o ebsa110.o errors.o \
   models.o pagetab.o profiler.o tracer.o trickbox.o \
   validate.o watchpnt.o winglass.o switch.o intc.o timer.o \
   watchdog.o dcc.o tube.o stackuse.o cnffile.o \
   example.o
   ```
5. Find the comment line:

   ```
   # Generated dependencies
   ```
6. Add all the dependencies, and the make rule, for your new file:

   ```
   example.o: $(SRCDIR1)/example.c
   example.o: $(SRCDIR1)/armdefs.h
   example.o: $(SRCDIR1)/rdi_hif.h
              $(CC) $(CFLAGS) $(CFLexample) -o example.o \
              $(SRCDIR1)/example.c
   ```
7. Type:

   `make -f` *`makefile`*.

3.4.2 Rebuilding on Windows

To rebuild `armulate.dll`, you need to have Microsoft Visual C++ version 5.0 installed.

1. Double click on the file *`Install_Directory`*`\ARMulate\Win32\ARMulate\dll\armulate.dsp`.

   Microsoft Developer Studio will open the project.

2. Add your new file or files to the project.

3. To rebuild `armulate.dll`, click on **Build→Build armulate.dll**.

4. Back up the old `armulate.dll` from *`Install_Directory`*`\bin`, and then copy your new `armulate.dll` file into *`Install_Directory`*`\bin`.

Alternatively, in the *`Install_Directory`*`\ARMulate\Win32\ARMulate\dll` directory, you can type:

```
nmake /f armulate.mak CFG="armulate - Win32 release"
```

at the DOS command prompt.

3.4.3 Rebuilding the ARM966E-S

To emulate the ARM966E-S core, you must use a special version of ARMulator, supplied in file `armul9xxe.dll`.

You can rebuild the `armul9xxe.dll` ARMulator in a similar manner to rebuilding the standard `armulate.dll` ARMulator.

There is a separate visual C++ project.

The rebuild kit (`armulate\Win32\`) has two directories:

- `armul9xxE`
- `arm9tdmi`

The dsp file (`armulate9xxE`) and makefile (`armulate9xxE.mak`) are in `armul9xxE\dll\`.

The models are in the directory `armul9xxE/user/`.

3.4.4 Giving a rebuilt ARMulator a different name

You can give your new version of the ARMulator a different name. We recommend that you use the same name that you used for the copy of the `ARMulate` directory.

If you do this, you can choose whichever version you want when choosing your target from the debugger. See *ADS Debuggers Guide* for information about selecting targets.

To give your ARMulator a different name, do the following:

1. Find the `armu8dll.def` file. In the line that reads:

   ```
   LIBRARY ARMULATE
   ```

   and replace `ARMULATE` with the name of your new armulator.

2. Change the name of your new `armulate.dll` before moving it into *`Install_Directory`*.

3.5 Configuring ARMulator to use the example

By default, ARMulator determines which memory model to use by reading the configuration file, `armul.cnf` (you can make ARMulator use a different configuration file, with a different name, by editing `cnffile.c`). See also *armul.cnf, the ARMulator configuration file* on page 4-98.

Before the example memory model can be used by ARMulator, a reference to it must be added to the configuration file. By default, ARMulator uses the built-in `Flat` or `MapFile` memory models.

Follow these steps to edit the configuration file so that ARMulator selects the sample memory model:

1. Load the `armul.cnf` file into a text editor, and find the following lines approximately halfway through the file:

   ```
   ;; List of memory models
   { Memories

   ;; the 'default' default is the flat memory map
   Default=Flat
   ```

2. Change the last two lines to:

   ```
   ;; Use the new memory model instead
   Default=Example
   ```

   The changed lines specify that the default memory model is now `Example`, rather than `Flat`.

   Note

   If a map file exists (or for ADW, if a map file is specified), the `armmap` memory model is used.

3. Start AXD, ADU, ADW, or armsd. The debugger responds:

   ```
   ARMulator 2.10
   ARM7, User manual example, 1MB memory, Dummy MMU,
   Soft Angel 1.4 [Angel SWIs], Profiler, Tracer, Pagetables,
   Big endian.
   ```

   You may see the following errors:

   - The *Floating-point Emulator* (FPE) initialization failed because this model does not have a standard memory map, and the FPE could not be loaded.
   - Alternatively, you might see the error:

     ```
     Initialization failed: Memory model 'Example'
     incompatible with bus interface
     ```

This means that the memory model cannot communicate with the selected processor (for example, ARM7TDMI, or ARM9TDMI).

Chapter 4
ARMulator Reference

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

4.1 ARMulator models

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

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

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 a configuration file (`armul.cnf`) during initialization, see *ToolConf* on page 4-108.

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

4.1.2 The ARMul_State state pointer

`ARMul_State` is an opaque data type that is a handle to the internal state of ARMulator. All the models are passed a *`state`* variable of type `ARMul_State`. ARMulator exports a number of functions to enable you to access ARMulator state. See *Accessing ARMulator state* on page 4-41 for more information.

4.2 Basic model interface

The simplest model interface is the basic model. This provides a mechanism for calling a user-supplied function during initialization (see *Basic model initialization function* on page 4-7). The function can then install upcalls, for example, to add functionality.

Basic models can be initialized either before or after memory models are initialized. This means that there are two distinct types of basic model:

- early models
- late models.

Whether a basic model is early or late is controlled by the location of its configuration in the configuration file, see *armul.cnf, the ARMulator configuration file* on page 4-98.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.2.1 Early basic models

Early basic models are initialized before memory models and can change the way the memory interface is initialized, primarily through calling `ARMul_InstallMemoryInterface()` (see *ARMul_InstallMemoryInterface* on page 4-8). In particular, early basic models can be used to install additional, *veneer* memory models.

Early models must not call the memory system, for example `ARMul_WriteWord()`, because it is not initialized when the early model is called.

The `watchpnt.c` and `tracer.c` models are examples of early basic models. These models install watchpoint and trace veneer memory models. The following sections give more information on installing a veneer memory model.

Installing a veneer memory model

By default, the ARMulator initialization sequence installs the default memory model for a specific processor core. For example, Figure 4-1 shows the model hierarchy after the memory model initialization function has completed.

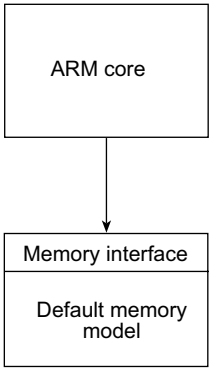

Figure 4-1 Minimal ARMulator model

You can use an early basic model to install any number of veneer memory models. The sequence of events is:

1. Define the early model in `armul.cnf`. ARMulator calls the initialization function for the early model (see *Basic model initialization function* on page 4-7).
2. The early model initialization function must call `ARMul_InstallMemoryInterface()` to install the memory interface for the veneer memory model. This is required only if you are installing veneer memory models (see *ARMul_InstallMemoryInterface* on page 4-8).
3. When the initialization function for the early model returns, ARMulator calls the memory model initialization function for the veneer memory model (see *Memory model initialization function* on page 4-15).

   The initialization function must call the initialization function for the model underneath it, either another veneer model or the standard memory model if there are no more veneer memory models installed.
4. When all veneer models are installed, the initialization function for the standard memory model for the processor model is called, see *Memory model initialization function* on page 4-15.

Figure 4-2 shows an example of a model hierarchy with the watchpoint veneer installed.

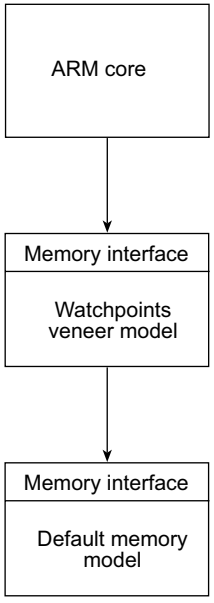

Figure 4-2 Veneer model hierarchy

4.2.2 Late basic models

Late basic models are initialized after the memory models. They can call the memory system, and can, for example, initialize the memory contents. The `pagetable.c` model is an example of a late basic model. It writes an MMU pagetable to memory, after the memory system and MMU have been initialized.

4.2.3 Basic model initialization function

A basic model exports a function that is called during initialization. You must provide the model initialization function. If the model and the function are registered, and an `armul.cnf` entry is found, then the model initialization function is called.

The name of the function is defined by you. In the description below, the name `ModelInit` is used.

Syntax

```
static ARMul_Error ModelInit(ARMul_State *state,
                             toolconf config)
```

where:

state is the ARMulator state pointer.

config is the configuration database.

Return

This function returns either:

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

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 4-77).

Example

The following example is from `watchpnt.c`:

```
#define ModelName (tag_t)"WatchPoints"

static ARMul_Error ModelInit(ARMul_State *state,
                             toolconf config)
{
    return ARMul_InstallMemoryInterface(state, TRUE, ModelName);
}

ARMul_ModelStub ARMul_WatchPointsInit = {
    ModelInit,
    ModelName
};
```

4.2.4 ARMul_InstallMemoryInterface

This function must be called from an early basic model that is installing a veneer memory model. It installs the memory interface for the veneer memory model.

Syntax

```
ARMul_Error ARMul_InstallMemoryInterface(ARMul_State *state,
                                         unsigned at_core,
                                         tag_t new_model)
```

where:

state is a pointer to the ARMulator state.

at_core indicates where to place the model:

- FALSE places the model immediately above the lowest memory model in the memory hierarchy.
- TRUE places the model immediately below the processor.

new_model names the veneer memory model.

Return

This function returns either:

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

See armerrs.h and errors.h for a full list of error codes. The error should be passed through ARMul_RaiseError() for formatting (see *ARMul_RaiseError* on page 4-77).

Usage

This function must be called before the memory models are initialized, for example, from an early model (see *Early basic models* on page 4-4).

For a simple processor and memory system, *at_core* has no effect, because the lowest memory model is the one immediately below the processor. However, for a cached processor, a cache model sits between the processor and the lowest memory model, as shown in Figure 4-3 on page 4-9.

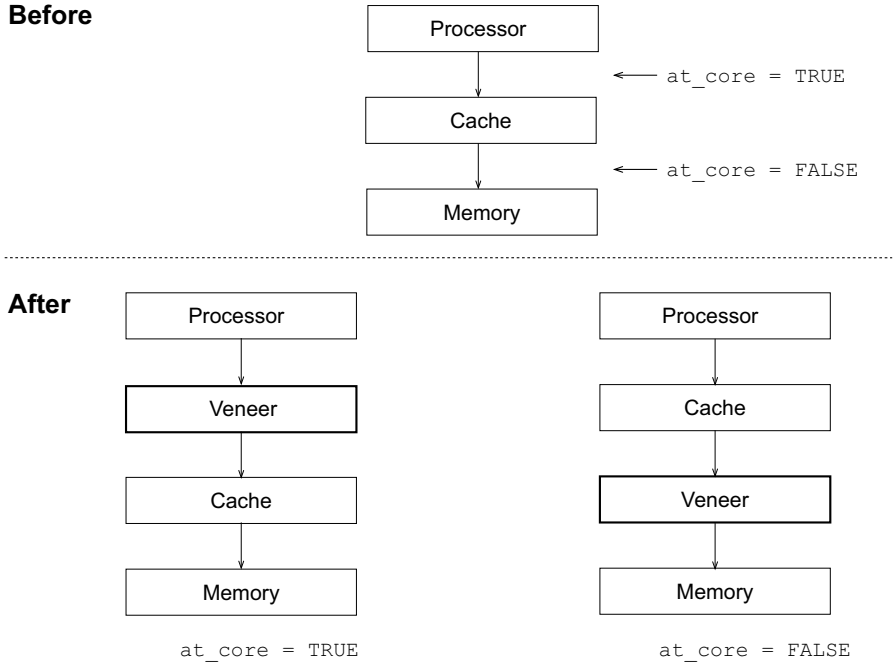

Figure 4-3 Inserting into a cache hierarchy

4.3 The memory interface

The memory interface is the interface between the ARMulator core and the memory model.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

Because there are many core processor types, there are many memory type variants. The memory initialization function is told which type it should provide (see *Memory model initialization function* on page 4-15). A model must refuse to initialize in the case of an unrecognized memory type variant.

If you install a veneer memory model between the default memory model and the ARM core, you must explicitly install the memory interface for the veneer model by calling `ARMul_InstallMemoryInterface()`. See *ARMul_InstallMemoryInterface* on page 4-8 and *Installing a veneer memory model* on page 4-4 for more information.

Note

The **nTRANS** signal from the processor is not passed to the memory interface. Because this signal changes infrequently and might not be used by a memory model, a model should use `TransChangeUpcall()` to track **nTRANS** (see *TransChangeUpcall* on page 4-57).

4.3.1 Memory type variants

The memory type variants are defined in the `ARMul_MemInterface` structure in `armmem.h`. They are described in the following sections:

- *Basic memory types* on page 4-11
- *Cached versions of basic memory types* on page 4-12
- *Byte-lane memory for StrongARM* on page 4-12
- *ARM8 memory type* on page 4-13
- *ARM9 memory type* on page 4-13.

Basic memory types

There are three basic variants of memory type. All three use the same function interface to the core. The types are defined as follows:

`ARMul_MemType_Basic`
: supports byte and word loads and stores.

`ARMul_MemType_16Bit`
: is the same as `ARMul_MemType_Basic` but with the addition of halfword loads and stores.

`ARMul_MemType_Thumb`
: is the same as `ARMul_MemType_16Bit` but with halfword instruction fetches (that can be sequential). This can indicate to a memory model that *most* accesses will be halfword-instruction-sequential rather than the usual word-instruction-sequential.

Note

Memory models that do not support halfword accesses should refuse to initialize for `ARMul_MemType_16Bit` and `ARMul_MemType_Thumb`.

For all three types, the model should fill in the `interf->x.basic` function pointers.

The file `armflat.c` contains an example function that implements a basic model.

Cached versions of basic memory types

There are three variants of the basic memory types for cached processors such as the ARM710 and ARM740T. These variants are defined as follows:

- `ARMul_MemType_BasicCached`
- `ARMul_MemType_16BitCached`
- `ARMul_MemType_ThumbCached`.

These differ from the basic equivalents in that there are only two types of cycle:

- Memory cycle, where `acc_MREQ(acc)` is `TRUE`
- Idle cycle, where `acc_MREQ(acc)` is `FALSE`.

A nonsequential access consists of an Idle cycle followed by a Memory cycle, with the same `address` supplied for both.

A sequential access is a Memory cycle, with `address` incremented from the previous access.

Byte-lane memory for StrongARM

StrongARM variants are defined as follows:

- `ARMul_MemType_StrongARM`
- `ARMul_MemType_ByteLanes`.

Externally, StrongARM can use a byte-lane memory interface. There is a StrongARM variant of the basic memory type that handles this. All the function types are the same, and the model must still fill in the basic part of the `ARMul_MemInterface` structure, but the meaning of the `ARMul_acc` word passed to the `access()` function is different.

The StrongARM variant replaces `acc_WIDTH` (see *armul_MemAccess* on page 4-20) with `acc_BYTELANE(acc)`. This returns a four-bit mask of the bytes in the word passed to the `access()` function that are valid.

There is no byte-order problem with this method of access. The model can ignore byte order. Bit 0 of this word corresponds to bits 0-7 of the data, bit 1 to bits 8-15, bit 2 to bits 16-23, and bit 4 to bits 24-31.

Note

Byte-lane memory for ARM7TDMI is not supported.

ARM8 memory type

The ARM8 memory type is defined as:

```
ARMul_MemType_ARM8
```

This is a double bandwidth interface. The ARM8 core can request two sequential accesses per cycle.

ARM9 memory type

The ARM9 memory type is defined as:

```
ARMul_MemType_ARM9
```

4.4 Memory model interface

The memory model interface is defined in the file `armmem.h` (which is `#included` from `armdefs.h`). All memory accesses are performed through a single function pointer that is passed a flags word. The flags word consists of a bitfield in which the bits correspond to the signals on the outside of the ARM processor. This determines the type of memory access that is being performed.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

At initialization time, the initialization function registers a number of functions in the memory interface structure, `ARMul_MemInterface` in `armmem.h`. The basic entries are:

```
typedef struct armul_meminterface ARMulMemInterface;
struct armul_meminterface {
    void *handle;
    armul_ReadClock *read_clock;
    armul_ReadCycles *read_cycles;
    union {
        struct {
            armul_MemAccess *access;
            armul_GetCycleLength *get_cycle_length;
        } basic;
        // ... other processor specific entries follow
```

The following sections describe the initialization function and the basic function entries:

- *Memory model initialization function* on page 4-15
- *armul_ReadClock* on page 4-17
- *armul_GetCycleLength* on page 4-17
- *armul_ReadCycles* on page 4-18
- *armul_MemAccess* on page 4-20.

There are two functions that allow you to set and return the address of the top of memory. These are described in:

- *ARMul_SetMemSize* on page 4-22
- *ARMul_GetMemSize* on page 4-22.

4.4.1 Memory model initialization function

The memory model exports a function that is called during initialization. You must provide the memory model initialization function. If the model and the function are registered, and an `armul.cnf` entry is found, then the memory model initialization function is called.

The name of the function is defined by you. In the description below, the name `MemInit` is used.

Syntax

```
static ARMul_Error MemInit(ARMul_State *state,
                           ARMul_MemInterface *interf,
                           ARMul_MemType variant,
                           toolconf config)
```

where:

state is a pointer to the ARMulator state.

interf is a pointer to the memory interface structure. See the `ARMul_MemInterface` structure in `armmem.h` for an example.

variant is the memory interface variant. See the `ARMul_MemType` enumeration in `armmem.h`. Refer to *Memory type variants* on page 4-11 for a description of the variants.

config is the configuration database.

Return

This function returns either:

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

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 4-77).

Usage

The initialization should set the handle for the model by assigning to `interf->handle`. The handle is usually a pointer to the state representing this instantiation of the model. It is passed to all the access functions called by ARMulator.

This function should also be used to:

- register any upcalls
- announce itself to the user using `ARMul_PrettyPrint()`
- attach any associated coprocessor models (CP15, for example) and set up its state.

Example

Refer to the definition of `MemInit` in `armflat.c` for an example. `MemInit` installs `ReadClock()`, `ReadCycles()`, `MemAccess()`, and `GetCycleLength()` functions. Refer to the following sections for more information on implementing these functions:

- *armul_ReadClock* on page 4-17
- *armul_GetCycleLength* on page 4-17
- *armul_ReadCycles* on page 4-18
- *armul_MemAccess* on page 4-20.

4.4.2 armul_ReadClock

This function should return the elapsed time in µ-seconds since the simulation model reset.

The `read_clock` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_ReadClock()` function.

Syntax

```
unsigned long armul_ReadClock(void *handle)
```

where:

handle is the value of `interf->handle` set in `MemInit`.

Return

The function returns an unsigned long value representing the elapsed time in µ-seconds since the model reset.

Usage

A model can supply NULL if it does not support this functionality.

4.4.3 armul_GetCycleLength

The `get_cycle_length` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_GetCycleLength()` function. This function should return the length of a single cycle in units of one tenth of a nanosecond.

You should implement this function, even if the implementation is very simple. The function name is defined by you.

Syntax

```
unsigned long armul_GetCycleLength(void *handle)
```

where:

handle is the value of `interf->handle` set in `MemInit`.

Return

The function returns an unsigned long representing the length of a single cycle in units of one tenth of a nanosecond. For example, it returns 300 for a 33.3MHz clock.

4.4.4 armul_ReadCycles

The `read_cycles` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_ReadCycles()` function. This function should calculate the total cycle count since the simulation model reset. You should implement this function, even if the implementation is very simple. The function name is defined by you.

Syntax

```
const ARMul_Cycles *armul_ReadCycles(void *handle)
```

where:

handle is the value of `interf->handle` set in `MemInit`.

Return

The function is called each time the counters are read by the debugger. The function calculates the total cycle count and returns a pointer to the `ARMul_Cycles` structure that contains the cycle counts. The `ARMul_Cycles` structure is defined as:

```
typedef struct {
          unsigned long Total;
          unsigned long NumNcycles, NumScycles,
                        NumCcycles, NumIcycles, NumFcycles;
          unsigned long CoreCycles;
} ARMul_Cycles;
```

Usage

A model can keep count of the accesses made to it by ARMulator by providing this function. The value of the `CoreCycles` field in `ARMul_Cycles`, is provided by ARMulator, not the memory model. When you write this function you must calculate the `Total` field, because this is the value returned when `ARMul_Time()` is called. See *Event scheduling functions* on page 4-67 for a description of `ARMul_Time()`.

These counters are also used to provide the `$statistics` variable inside the ARM debuggers, if the memory model does not use `ARMul_AddCounterDesc()` and `ARMul_AddCounterValue()`. (see *Upcalls* and *General purpose functions* on page 4-77).

Example

```
ARMul_Cycles *cycles;
cycles = interf->read_cycles(handle);
// where interf is a pointer to the memory interface structure.
// and handle is a void * pointer to the ARMul_State structure.
```

4.4.5 armul_MemAccess

The `access` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_MemAccess()` function. This function is called on each ARM core cycle.

You must implement this function, even if the implementation is very simple. The function name is defined by you.

Syntax

```
int armul_MemAccess(void *handle, ARMword address,
                    ARMword *data, ARMul_acc access_type)
```

where:

handle is the value assigned to `interf->handle` in the initialization function.

address is the value on the address bus.

data is a pointer to the data for the memory access. See *Usage* below for details.

access_type

encodes the type of cycle. On some processors (for example, cached processors) some of the signals will not be valid. The macros for determining access type are:

`acc_MREQ(acc)`

chooses between memory request and non-memory request accesses.

`acc_WRITE(acc)`

`acc_READ(acc)`

for memory cycles, determines whether this is a read or write cycle (not `acc_READ` implies `acc_WRITE`, and not `acc_WRITE` implies `acc_READ`).

`acc_SEQ(acc)`

for a memory cycle, this is `TRUE` if the address is the same as, or sequentially follows from the address of the preceding cycle. For a non-memory cycle it distinguishes between coprocessor (`acc_SEQ`) and idle (not `acc_SEQ`) cycles.

`acc_OPC(acc)`

for memory cycles, this is `TRUE` if the data being read is an instruction. (It is never `TRUE` for writes.)

`acc_LOCK(acc)`
: distinguishes a read-lock-write memory cycle.

`acc_ACCOUNT(acc)`
: is `TRUE` if the cycle is coming from the ARM core, rather than the remote debug interface.

`acc_WIDTH(acc)`
: returns `BITS_8`, `BITS_16`, or `BITS_32` depending on whether a byte, halfword, or word is being accessed.

Return

The function returns:

- 1, to indicate successful completion of the cycle
- 0, to tell the processor to busy-wait and try the access again next cycle
- –1, to signal an abort
- –2, to indicate that an address was not decoded by a peripheral model (see *Reference peripherals* on page 4-121).

Usage

Reads For reads, the memory model function should write the value to be read by the core to the word pointed to by *`data`*. For example, with a byte load it should write the byte value, with a halfword load it should write the halfword value.

The model can ignore the alignment of the address passed to it because this is handled by ARMulator. However, it must present the bytes of the word in the correct order for the byte order of the processor. This can be determined by using either a `ConfigChangeUpcall()` upcall or `ARMul_SetConfig()` (see *Accessing ARMulator state* on page 4-41).

`armdefs.h` provides a flag variable macro named `HostEndian`, which is `TRUE` if ARMulator is running on a big-endian machine. See the `armflat.c` memory model for an example of how to handle byte order.

Writes For writes, *`data`* points to the datum to be stored. However, this value may need to be shortened for a byte or halfword store.

As with reads, byte order must be handled correctly.

4.4.6 ARMul_SetMemSize

This function can be called during memory initialization. It specifies the size, and therefore the top of memory.

Syntax

```
ARMword ARMul_SetMemSize(ARMul_State *state, ARMword size)
```

where:

state is a pointer to the ARMulator state.

size is the size of memory in bytes (word aligned).

Return

The function returns the previous `MemSize` value.

Usage

The value of `size` should not exceed 0x80000000.

4.4.7 ARMul_GetMemSize

This function returns the address of the top of memory.

Syntax

```
ARMword ARMul_GetMemSize(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the highest available address in memory.

Usage

This function can be used, for example, by a debug monitor model to tell an application where the top of usable memory is, so it can set up application memory.

4.5 Coprocessor model interface

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

- *init* on page 4-27
- *ldc* on page 4-28
- *stc* on page 4-29
- *mrc* on page 4-30
- *mcr* on page 4-31
- *cdp* on page 4-32.

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 4-33
- *write* on page 4-34.

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

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.5.1 The ARMul_CPInterface structure

The coprocessor initialization structure contains a set of function pointers for each supported operation. You must use the coprocessor initialization function to install your functions in the structure at initialization. Refer to *init* on page 4-27 for more information.

This structure also contains a pointer to a `reg_bytes` array that contains:

- the number of coprocessor registers, in the first element
- the number of bytes available to each register, in the remaining elements.

For example, `dummymmu.c` defines an array of eight registers, each of four bytes:

```
static const unsigned int MMURegBytes[] = {8, 4,4,4,4,4,4,4,4};
```

Definition

The `ARMul_CPInterface` structure is defined as:

```
typedef struct ARMul_cop_interface_str ARMul_CPInterface;

struct ARMul_cop_interface_str {
    void *handle;                    /* A model private handle */
    armul_LDC *ldc;                  /* LDC instruction */
    armul_STC *stc;                  /* STC instruction */
    armul_MRC *mrc;                  /* MRC instruction */
    armul_MCR *mcr;                  /* MCR instruction */
    armul_CDP *cdp;                  /* CDP instruction */
    armul_CPRead *read;              /* Read CP register */
    armul_CPWrite *write;            /* Write CP register */
    const unsigned int *reg_bytes;   /* map of CP reg sizes */
}
```

Example

In Example 4-1 on page 4-25, if the core has a *memory management unit* (MMU), a predefined `mmu->RegBytes` is used. If the core has a *protection unit* (PU), the size of `RegBytes[7]` and `RegBytes[8]` is modified. `RegBytes[7]` corresponds to CP register 6 and `RegBytes[8]` corresponds to CP register 7.

CP register 6 is the protection region base and size register and has eight indexable registers, so it is set to size `sizeof(ARMword)*8`.

Refer to *ARM Architecture Reference Manual* for more information on the MMU and PU.

Example 4-1

```
{
    if (mem->prop & Cache_ProtectionUnit_Prop)
        {
        /* Use the PU */
        mmu->RegBytes[0]=8;                         /* has 8 registers */
        mmu->RegBytes[7]=sizeof(ARMword)*8;         /* register 7 is 8 words long */
        mmu->RegBytes[8]=sizeof(ARMword);           /* register 8 is a single word */
        interf->mrc=PU_MRC;
        interf->mcr=PU_MCR;
        interf->read=PU_CPRead;
        interf->write=PU_CPWrite;
        interf->reg_bytes=mmu->RegBytes;
        ARMul_PrettyPrint(state,", PU");



        /* Initialise PU Area registers to 0 */
        for ( i=0; i<=7; i++)
            {
            mmu->PU_Areas[i].PU_Register=0;
            }
        }
    else /* Use the MMU */
        {
        interf->mrc=MRC;
        interf->mcr=MCR;
        interf->read=CPRead;
        interf->write=CPWrite;
        interf->reg_bytes=mmu->RegBytes;
        ARMul_PrettyPrint(state,", MMU");
        }
}
```

4.5.2 ARMul_CoProAttach

Coprocessors are either initialized directly by ARMulator as appropriate, or can be attached directly by another model by calling `ARMul_CoProAttach()`. As with memory models, the coprocessor initialization function is used to fill in the interface structure. `ARMul_CoProAttach()` registers the coprocessor initialization function for a specified coprocessor.

Syntax

```
ARMul_Error ARMul_CoProAttach(ARMul_State *state,
                              unsigned number,
                              const ARMul_CPInit *init,
                              toolconf config,
                              void *sibling)
```

where:

state is a pointer to the ARMulator state.

number is the coprocessor number to attach.

init is a pointer to a coprocessor initialization function.

config is the configuration database.

sibling is a pointer to the state to be shared with the coprocessor.

Return

This function returns either:

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

See `armerrs.h` and `errors.h` for a full list of error codes. The error should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 4-77).

Example

```
error = ARMul_CoProAttach(state, 4, init, config, handle);
```

4.5.3 init

This is the coprocessor initialization function. This function fills in the `ARMul_CPInterface` structure for the coprocessor model (see *The ARMul_CPInterface structure* on page 4-24).

Syntax

```
ARMul_Error init(ARMul_State *state, unsigned num,
                 ARMul_CPInterface *interf, toolconf config,
                 void *sibling)
```

where:

state is a pointer to the ARMulator state.

num is the coprocessor number.

interf is a pointer to the `ARMul_CPInterface` structure to be filled in.

config is the configuration database.

sibling identifies associations between the coprocessor and other simulated components, such as sibling coprocessors. For example, a system may have a pair of coprocessors that must be aware of each other. This is the value passed to `ARMul_CoProAttach()`.

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 should be passed through `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 4-77).

4.5.4 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 value of `interf->handle` set in `init`.

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 should 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`).

4.5.5 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 value of `interf->handle` set in `init`.

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 should 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 should 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`).

4.5.6 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 should return ARMul_CANT.

Syntax

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

where:

handle is the value of interf->handle set in init.

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 should 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.

4.5.7 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 should return ARMul_CANT.

Syntax

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

where:

handle is the value of interf->handle set in init.

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 should 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.

4.5.8 cdp

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

Syntax

```
unsigned cdp(void *handle, unsigned type, ARMword instr)
```

where:

handle is the value of interf->handle set in init.

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 should 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.

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.

4.5.9 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 should return `ARMul_CANT`.

Syntax

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

where:

handle is the value of `interf->handle` set in `init`.

reg is the register number of the coprocessor register to be read.

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.

4.5.10 write

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

Syntax

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

where:

handle is the value of `interf->handle` set in `init`.

reg is the register number of the coprocessor register that is to be written.

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`.

4.6 Operating system or debug monitor interface

ARMulator supports rapid prototyping of low-level operating system code through an interface that enables a model to intercept SWIs and exceptions, and model them on the host. This model can communicate with the simulated application by reading and writing the simulated ARM state using the routines described in *Accessing ARMulator state* on page 4-41.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

The interface functions are:

- *init* on page 4-36
- *handle_swi* on page 4-37
- *exception* on page 4-38.

These functions are described in more detail in the following sections.

4.6.1 The ARMul_OSInterface structure

The `ARMul_OSInterface` structure is defined as:

```
typedef struct armul_os_interface ARMul_OSInterface;

typedef ARMul_Error armul_OSInit(ARMul_State *state,
                                 ARMul_OSInterface *interf,
                                 toolconf config);
typedef unsigned armul_OSHandleSWI(void *handle,ARMword number);
typedef unsigned armul_OSException(void *handle, ARMword vector,
                                   ARMword pc);

struct armul_os_interface {
  void *handle;                    /* A model private handle */
  armul_OSHandleSWI *handle_swi;   /* SWI handler */
  armul_OSException *exception;    /* Exception handler */
};

typedef struct {
  armul_OSInit *init;              /* O/S initializer */
  tag_t name;                      /* O/S name */
} ARMul_OSStub;
```

4.6.2 init

This is the OS initialization function. It is passed a vector of functions to fill in. As with other models, the operating system model is called through an initialization function exported in a stub.

The memory system is guaranteed to be operating at this time, so the operating system can read and write to the simulated memory using the routines described in *Memory access functions* on page 4-65.

Syntax

```
typedef ARMul_Error init(ARMul_State *state,
                         ARMul_OSInterface *interf,
                         toolconf config)
```

where:

state is a pointer to the ARMulator state.

interf is a pointer to the OS interface structure.

config is the configuration database.

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 `ARMul_RaiseError()` for formatting (see *ARMul_RaiseError* on page 4-77).

Usage

This function can also run initialization code.

4.6.3 handle_swi

This is the OS model SWI handling function. It is called whenever a `SWI` instruction is executed. This enables support code to simulate operating system operations. This code can model as much of your operating system as you choose.

Syntax

```
typedef unsigned handle_swi(void *handle, ARMword number)
```

where:

handle is the value of `interf->handle` set in `init`.

number is the SWI number.

Return

The function can refuse to handle the `SWI` by returning `FALSE`, or the model may choose not to handle `SWI` instructions by setting NULL as the `handle_swi` function. In either case, the SWI exception vector is taken by ARMulator. If the function returns `TRUE` ARMulator continues from the next instruction after the `SWI`.

4.6.4 ARMulator SWIs

In addition to the standard Angel SWIs, ARMulator uses a set of SWIs for default exception vector handlers. These are known as the *soft vector SWIs*. The soft vector code is installed by the Angel model.

There are two sets of SWIs:

SWIs 0x90 – 0x98 are used to implement `$vector_catch`. That is, they return control to the debugger if the user has set `$vector_catch` for the relevant exception vector. SWI 0x90 is used for the reset vector, 0x91 for the undefined instruction vector, and so on.

SWIs 0x80 – 0x88 are used to stop ARMulator if the exception cannot be handled. The 0x80 SWIs are used as a final stop if the exception is not caught by such an exception handler.

Note

These SWIs are for internal use by ARMulator only.

4.6.5 exception

This is the OS model exception handling function. It is called whenever an exception occurs.

Syntax

```
typedef unsigned exception(void *handle, ARMword vector,
                           ARMword pc)
```

where:

handle is the value of `interf->handle` set in `init`.

vector contains the address of the vector about to be executed, for example:

- `0x00` Reset
- `0x04` Undefined Instruction
- `0x1C` Fast Interrupt (FIQ).

pc contains the program counter (including the effect of pipelining) at the time the exception occurred.

Return

If the function returns `TRUE`, ARMulator continues from the instruction following the instruction that was being executed when the exception occurred.

Note

If the exception is a Prefetch or Data Abort, the user function must make ARMulator retry the instruction, rather than continuing from the following instruction. The user function can set up the pc by calling `ARMul_SetPC` to ensure this, before returning `TRUE`.

A return value of `FALSE` causes ARMulator to handle the exception normally.

Usage

The CPU state is frozen immediately after the exception has occurred, but before the CPU has switched processor state or taken the appropriate exception vector.

4.7 Using the floating-point emulator

ARMulator is supplied with the *floating-point emulator* (FPE) in object form. If the FPE is selected on initialization, the debug monitor model (`angel.c`) loads and starts executing the FPE.

The FPE requires the following SWIs to be supported by the debug monitor. Angel does not support these SWIs, however they are implemented by `angel.c` to support FPE:

- `SWI_Exit` (0x11)
- `SWI_GenerateError` (0x71).

To load and initialize the FPE, call the following functions:

- `ARMul_FPEInstall()`
- `ARMul_FPEVersion()`
- `ARMul_FPEAddressInEmulator()`.

These are described in more detail in the following sections.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.7.1 ARMul_FPEInstall

This function writes the FPE into memory (below 0x8000), and executes it.

Syntax

```
int ARMul_FPEInstall(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Usage

Note

Because this involves running code, it must be done only after ARMulator is fully initialized. Before calling `ARMul_FPEInstall()`, Angel completely initializes itself.

Return

The function returns:

- `TRUE`, if the installation is successful
- `FALSE`, if the installation fails.

4.7.2 ARMul_FPEVersion

This function returns the FPE version number. Angel uses this for unwinding aborts inside the emulator (see the `angel.c` source code for details).

Syntax

```
int ARMul_FPEVersion(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns either:

- the FPE version code, if available
- –1 if there is no FPE.

4.7.3 ARMul_FPEAddressInEmulator

This function returns `TRUE` if the specified address lies inside the emulator.

Syntax

```
int ARMul_FPEAddressInEmulator(ARMul_State *state, ARMword addr)
```

where:

state is a pointer to the ARMulator state.

addr is the address to check.

Return

The function returns:

- `FALSE`, if there is no FPE, or the address is not in the FPE
- `TRUE`, if the address is in the FPE.

4.8 Accessing ARMulator state

All the models are passed a `state` variable of type `ARMul_State`. This is an opaque handle to the internal state of ARMulator. ARMulator exports these functions to enable models to access the ARMulator state through this handle.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

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

- *ARMul_GetMode* on page 4-42
- *ARMul_GetReg* on page 4-43
- *ARMul_SetReg* on page 4-44
- *ARMul_GetR15 and ARMul_GetPC* on page 4-45
- *ARMul_SetR15 and ARMul_SetPC* on page 4-45
- *ARMul_GetCPSR* on page 4-46
- *ARMul_SetCPSR* on page 4-46
- *ARMul_GetSPSR* on page 4-47
- *ARMul_SetSPSR* on page 4-47.

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

- *ARMul_CPRegBytes* on page 4-48
- *ARMul_CPRead* on page 4-48
- *ARMul_CPWrite* on page 4-49.

The following function enables you to change the configuration of your modeled processor:

- *ARMul_SetConfig* on page 4-50.

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 may be called during simulation of an instruction. In contrast, it is 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.

In addition, the special value `CURRENTMODE` is defined. This enables `ARMul_GetReg()` to return the current mode number.

Table 4-1 Defined processor modes

`USER32MODE`	`ABORT32MODE`
`FIQ32MODE`	`UNDEF32MODE`
`IRQ32MODE`	`SYSTEM32MODE`
`SCV32MODE`	

4.8.1 ARMul_GetMode

This function returns the current processor mode.

Syntax

```
ARMword ARMul_GetMode(ARMul_State *state)
```

where:

`state` is a pointer to the ARMulator state.

Returns

This function returns the current mode.

See Table 4-1 on page 4-42 for a list of defined processor modes.

Usage

If this is to be done frequently, a model should install a `ModeChange()` upcall instead (see *ModeChangeUpcall* on page 4-56).

4.8.2 ARMul_GetReg

This function reads a register for a specified processor mode.

Syntax

```
ARMword ARMul_GetReg(ARMul_State *state, unsigned mode,
                     unsigned reg)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode. Values for mode are defined in `armdefs.h` (see Table 4-1 on page 4-42).

reg is the number of the register to read.

Return

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

Usage

Note

Register r15 must not be accessed with this function. Use `ARMul_GetPC()` or `ARMul_GetR15()` as described in *ARMul_GetR15 and ARMul_GetPC* on page 4-45.

4.8.3 ARMul_SetReg

This function writes a register for a specified processor mode.

Syntax

```
void ARMul_SetReg(ARMul_State *state, unsigned mode,
                  unsigned reg, ARMword value)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode. Mode numbers are defined in `armdefs.h` (see Table 4-1 on page 4-42).

reg is the number of the register to write.

value is the value to be written to register `reg` for the specified processor mode.

Usage

Note

Register r15 must not be accessed with this function. Use `ARMul_SetPC()`, or `ARMul_SetR15()` as in *ARMul_SetR15 and ARMul_SetPC* on page 4-45.

4.8.4 ARMul_GetR15 and ARMul_GetPC

The following functions read register r15.

Syntax

```
ARMword ARMul_GetR15(ARMul_State *state)
ARMword ARMul_GetPC(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The functions return the value of register r15. The effect of either variant is the same.

4.8.5 ARMul_SetR15 and ARMul_SetPC

The following functions write register r15.

Syntax

```
void ARMul_SetR15(ARMul_State *state, ARMword value)
```

```
void ARMul_SetPC(ARMul_State *state, ARMword value)
```

where:

state is a pointer to the ARMulator state.

value the new value of r15 (pc) to be written.

Return

The functions write a value into register r15. The effect of either variant is the same.

4.8.6 ARMul_GetCPSR

This function reads the CPSR.

Syntax

```
ARMword ARMul_GetCPSR(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the value of the CPSR for the current mode.

4.8.7 ARMul_SetCPSR

This function writes a value to the CPSR for the current processor mode.

Syntax

```
void ARMul_SetCPSR(ARMul_State *state, ARMword value)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode. Values for mode are defined in `armdefs.h` (see Table 4-1 on page 4-42).

value is the value to be written to the CPSR for the current mode.

4.8.8 ARMul_GetSPSR

This function reads the SPSR for a specified processor mode.

Syntax

```
ARMword ARMul_GetSPSR(ARMul_State *state, ARMword mode)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode for the SPSR to be read.

Return

The function returns the value of the SPSR for the specified mode.

4.8.9 ARMul_SetSPSR

This function writes the SPSR for a specified processor mode.

Syntax

```
void ARMul_SetSPSR(ARMul_State *state, ARMword mode,
                   ARMword value)
```

where:

state is a pointer to the ARMulator state.

mode is the processor mode for the SPSR to be read. Values for *mode* are defined in `armdefs.h` (see Table 4-1 on page 4-42).

value is the new value to be written to the SPSR for the specified mode.

4.8.10 ARMul_CPRegBytes

This function returns the `reg_bytes[]` array for the specified coprocessor (see *The ARMul_CPInterface structure* on page 4-24 for details).

Syntax

```
unsigned int const *ARMul_CPRegBytes(ARMul_State *state,
                                     unsigned CPnum)
```

where:

state is a pointer to the ARMulator state.

CPnum is the coprocessor number to return the `reg_bytes[]` array for.

4.8.11 ARMul_CPRead

This function calls the `read` method for a coprocessor. It also intercepts calls to read the FPE emulated registers (see *Using the floating-point emulator* on page 4-39).

Syntax

```
unsigned ARMul_CPRead(void *handle, unsigned reg,
                      ARMword *value)
```

where:

handle is a pointer to the ARMulator state.

reg is the number of the coprocessor register to read from.

value is the address to write the register value to.

Return

The function must return:

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

4.8.12 ARMul_CPWrite

This function calls the `write` method for a coprocessor. It also intercepts calls to write the FPE emulated registers (see *Using the floating-point emulator* on page 4-39).

Syntax

```
unsigned ARMul_CPWrite(void *handle, unsigned reg,
                       ARMword const *value)
```

where:

handle is a pointer to the ARMulator state.

reg is the number of the coprocessor register to write to.

value is the address of the data to write to the coprocessor register.

Return

The function must return:

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

4.8.13 ARMul_SetConfig

This function changes the config value of the modeled processor. The config value represents the state of the configuration pins on the ARM core. See *Configuration bits and signals* on page 4-58 for details of bit to signal assignments.

Syntax

```
ARMword ARMul_SetConfig(ARMul_State *state, ARMword changed,
                        ARMword config)
```

where:

state is a pointer to the ARMulator state.

changed is a bitmask of the config bits to change.

config contains the new values of the bits to change.

Return

The function returns the previous config value.

Usage

Note

If a bit is cleared in *changed* it must not be set in *config*. For example, to set bit 1 and clear bit 0:

changed 0x03 (00000011 binary)

config 0x02 (00000010 binary)

`ConfigChangeUpcall()` is called. See *ConfigChangeUpcall* on page 4-58 for more information on this upcall.

Example

```
oldConfig = ARMul_SetConfig(state, 0x00000001, 0x00000001);
// This sets bit 0 to value 1
oldConfig = ARMul_SetConfig(state, 0x00000002, 0x00000001);
// This sets bit 1 to value 0 - note that bit 0 is unaffected.
```

The following call can be used to obtain the current settings of the configuration pins, without modifying them:

```
currentConfig = ARMul_SetConfig(state, 0, 0);
```


4.9 Exceptions

The following functions enable a model to set or clear interrupts and resets, or branch to a SWI handler:

- *ARMul_SetNirq and ARMul_SetNfiq*
- *ARMul_SetNreset* on page 4-52
- *ARMul_SWIHandler* on page 4-52.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.9.1 ARMul_SetNirq and ARMul_SetNfiq

The following functions are used to set and clear IRQ and FIQ interrupts.

Syntax

```
unsigned ARMul_SetNirq(ARMul_State *state, unsigned value)
unsigned ARMul_SetNfiq(ARMul_State *state, unsigned value)
```

where:

state is a pointer to the ARMulator state.

value is the new `Nirq` or `Nfiq` signal value.

Note

The signals are active LOW:

- 0 = interrupt
- 1 = no interrupt.

Return

The functions return the old signal value.

Note

For information about signalling interrupts when using an interrupt controller see *Interrupt controller* on page 4-121.

4.9.2 ARMul_SetNreset

This function sets and clears RESET exceptions.

Syntax

```
unsigned ARMul_SetNreset(ARMul_State *state, unsigned value)
```

where:

state is a pointer to the ARMulator state.

value is the new Nreset signal value.

Note

The signal is active LOW:

- 0 = reset
- 1 = no reset.

Return

The function returns the old signal value.

4.9.3 ARMul_SWIHandler

This function can be called from a `handle_swi()` function to enter a SWI handler at a given address. It causes the processor to act as if it had taken the SWI vector, decoded the SWI number, and then branched to this address.

Syntax

```
void ARMul_SWIHandler(ARMul_State *state, ARMword address)
```

where:

state is a pointer to the ARMulator state.

address is the address of the instruction to branch to.

Usage

See the code for handling `SWI_GenerateError` in `angel.c`. for an example of how to use this function.

4.10 Upcalls

ARMulator can be made to call back your model when some state values change. You do this by installing the relevant *upcall*. In the context of ARMulator, the term upcall is synonymous with callback.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

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

You can use upcalls to avoid having to check state values on every access. For example, a memory 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 memory model can attach to the `ConfigChangeUpcall()` upcall to be informed when this signal changes.

Every upcall is called when ARMulator resets and after ARMulator initialization is complete, regardless of whether the signals have changed, with the exception of `UnkRDIInfoUpcall()` and `EventUpcall()`.

The upcalls are defined in `armdefs.h`. The following upcalls are described in the sections below:

- *ExitUpcall* on page 4-55
- *ModeChangeUpcall* on page 4-56
- *TransChangeUpcall* on page 4-57
- *ConfigChangeUpcall* on page 4-58
- *InterruptUpcall* on page 4-60
- *ExceptionUpcall* on page 4-61
- *UnkRDIInfoUpcall* on page 4-62
- *EventUpcall* on page 4-64.

Refer to *Installing an upcall* and *Removing an upcall* on page 4-54 for information on how to install and remove the upcalls.

4.10.1 Installing an upcall

Each upcall is installed using a function of the form:

```
void *ARMul_Install<UpcallName>(ARMul_State *state,
                                typename *fn,
                                void *handle)
```

where:

<UpcallName>
: is the name of the upcall. For example, the `ExitUpcall()` is installed with `ARMul_InstallExitHandler`.

state
: is a pointer to the ARMulator state.

typename
: is the type of the function, as defined by **typedef** in the upcall prototype.

fn
: is a pointer to the function to be installed.

handle
: is the handle to be passed to the upcall function.

The function returns a **void** * handle to the upcall handler. This must be kept because it is required by the corresponding `Remove` upcall function.

4.10.2 Removing an upcall

Each upcall is removed using a function of the form:

```
int *ARMul_Remove<UpcallName>(ARMulState *state, void *node)
```

where:

<UpcallName>
: is the name of the upcall to be removed. For example, the `ExitUpcall()` is removed with `ARMul_RemoveExitHandler`.

state
: is the state pointer.

node
: is the handle returned from the corresponding `Install` upcall function.

The remove upcall functions return:

- `TRUE` if the upcall is removed
- `FALSE` if the upcall remove failed.

4.10.3 ExitUpcall

The exit upcall is called when ARMulator exits. It should be used to release any memory used.

Syntax

```
typedef void armul_ExitUpcall(void *handle)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

Usage

Note

The ANSI `free()` function is a valid `ExitUpcall()`. If no exit upcall is registered and a model uses some memory, that memory will be lost.

Install the upcall using:

```
void *ARMul_InstallExitHandler(ARMul_State *state,
                               armul_ExitUpcall *fn,
                               void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveExitHandler(ARMul_State *state, void *node)
```

Refer to *Installing an upcall* on page 4-54 for more information.

4.10.4 ModeChangeUpcall

The mode change upcall is called whenever ARMulator changes mode. The upcall is passed both the *old* and *new* modes.

Syntax

```
typedef void armul_ModeChangeUpcall(void *handle, ARMword old,
                                    ARMword new)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

old is the old processor mode. Values for mode are defined in `armdefs.h` (see Table 4-1 on page 4-42).

new is the new processor mode. Values for mode are defined in `armdefs.h` (see Table 4-1 on page 4-42).

Usage

Install the mode change upcall using:

```
void *ARMul_InstallModeChangeHandler(ARMul_State *state,
                                     armul_ModeChangeUpcall *fn,
                                     void *handle)
```

Remove the mode change upcall using:

```
int ARMul_RemoveModeChangeHandler(ARMul_State *state,
                                  void *node)
```

Refer to *Installing an upcall* on page 4-54 for more information.

4.10.5 TransChangeUpcall

This upcall is called when the **nTRANS** signal on the ARM core changes.

The **nTRANS** signal is the Not Memory Translate signal. When LOW, it indicates that the processor is in User mode, or that the processor is executing an `LDRT`/`STRT` instruction from a non-User mode. It can be used to tell memory management models when translation of the addresses should be turned on, or as an indicator of non-User mode activity (for example, to provide different levels of access in non-User modes).

Refer to *ARM Architecture Reference Manual* for details of the `LDRT`/`STRT` instructions.

Syntax

```
typedef void armul_TransChangeUpcall(void *handle, unsigned old,
                                     unsigned new)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

old is the old **nTRANS** signal value.

new is the new **nTRANS** signal value.

Usage

Install the upcall using:

```
void *ARMul_InstallTransChangeHandler(ARMul_State *state,
                                      armul_TransChangeUpcall *fn,
                                      void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveTransChangeHandler(ARMul_State *state,
                                   void *node)
```

Refer to *Installing an upcall* on page 4-54 for more information.

4.10.6 ConfigChangeUpcall

This upcall is made when the ARMulator model configuration is changed (for example, from big-endian to little-endian). You can call `ARMul_SetConfig()` to change the configuration yourself (see *ARMul_SetConfig* on page 4-50).

Configuration is specified as a bitfield of config bits. The config bits represent signals to the configuration pins on the ARM core. Table 4-2 lists the bits that correspond to each signal.

Refer to *ARM Architecture Reference Manual* for more information on configuration signals.

If you have a CP15 then the control register bits corresponding to the signals listed in Table 4-2 will be set in the same way.

Table 4-2 Configuration bits and signals

Signal	Bit	Notes
ARMul_Prog32	bit 4	Always high on ARM7TDMI, ARM9TDMI
ARMul_Data32	bit 5	Always high on ARM7TDMI, ARM9TDMI
ARMul_LateAbt	bit 6	Not on ARM7, ARM9, or StrongARM
ARMul_BigEnd	bit 7	—
ARMul_BranchPredict	bit 11	ARM8 only

Syntax

```
typedef void armul_ConfigChangeUpcall(void *handle, ARMword old,
                                      ARMword new)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

old is a bitfield representing the old configuration.

new is a bitfield representing the new configuration.

Usage

Install the upcall using:

```
void *ARMul_InstallConfigChangeHandler(ARMul_State *state,
                                       armul_ConfigChangeUpcall *fn,
                                       void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveConfigChangeHandler(ARMul_State *state,
                                    void *node)
```

Refer to *Installing an upcall* on page 4-54 for more information.

4.10.7 InterruptUpcall

This upcall is called whenever the ARM core *notices* an interrupt (not when it takes an interrupt) or reset. It is called even if interrupts are disabled.

Syntax

```
typedef unsigned int armul_InterruptUpcall(void *handle,
                                           unsigned int which)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

which is a bitfield that encodes which interrupt(s) have been noticed:

- `bit 0` Fast interrupt request (FIQ).
- `bit 1` Interrupt request (IRQ).
- `bit 2` Reset.

Usage

This upcall can be used by a memory model to reset its state or implement a wake-up, for example. It is called at the start of the instruction or cycle (depending on the core being simulated) when the interrupt is noticed.

The interrupt responsible can be removed using `ARMul_SetNirq()` or `ARMul_SetNfiq()`, in which case the ARM will not notice the interrupt. See *ARMul_SetNirq and ARMul_SetNfiq* on page 4-51 for more information.

Note

You can use `ARMul_SetNirq()` and `ARMul_SetNfiq()` to clear the interrupt signal, but they will *not* necessarily clear the interrupt source itself.

Install the interrupt upcall using:

```
void *ARMul_InstallInterruptHandler(ARMul_State *state,
                    armul_InterruptUpcall *fn, void *handle)
```

Remove the interrupt upcall using:

```
int ARMul_RemoveInterruptHandler(ARMul_State *state, void *node)
```

Refer to *Installing an upcall* on page 4-54 for more information.

4.10.8 ExceptionUpcall

This upcall is called whenever the ARM processor takes an exception.

Syntax

```
typedef unsigned int armul_ExceptionUpcall(void *handle,
                                           ARMword vector,
                                           ARMword pc,
                                           ARMword instr)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

vector is the address of the appropriate hardware vector to be taken for the exception.

pc is the value of pc at the time the exception occurs.

instr 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 upcall returns nonzero, the ARM does not take the exception (the exception is ignored).

Note

In this release of ARMulator, this occurs in addition to the calling of the `handle_swi()` function of the installed operating system model. Future releases may not support the operating system interface, and you should use this upcall in preference. The model can be installed as a basic model (see *Basic model interface* on page 4-4). The models, such as `angel.c` and `validate.c`, shipped with this release of ARMulator can be built either as a basic model or as an operating system model.

Note

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

Install the exception upcall using:

```
void *ARMul_InstallExceptionHandler(ARMul_State *state,
                                    armul_ExceptionUpcall *fn,
                                    void *handle)
```

Remove the exception upcall using:

```
int ARMul_RemoveExceptionHandler(ARMul_State *state, void *node)
```

Refer to *Installing an upcall* on page 4-54 for more information.

4.10.9 UnkRDIInfoUpcall

`UnkRDIInfoUpcall()` functions are called if ARMulator cannot handle an `RDI_Info` request itself. They return an `RDIError` value. The `UnkRDIInfoUpcall()` 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 armul_UnkRDIInfoUpcall(void *handle, unsigned type,
                                   ARMword *arg1,
                                   ARMword *arg2)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

type is the `RDI_Info` subcode. These are defined in `rdi_info.h`. See below for some examples.

arg1/`arg2` are arguments passed to the upcall from the calling function.

Usage

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

The following codes are examples of the `RDI_Info` 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_Points`

`watchpnt.c` intercepts `RDIInfo_Points` to tell the debugger that ARMulator supports watchpoints. This is similar to the use of `RDIInfo_Target` in `profiler.c`.

`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 4-77) 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 4-77) 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 should always respond to these messages with `RDIError_UnimplementedMessage`, so that the message is passed on even if the model has responded.

Install the upcalls using:

```
void *ARMul_InstallUnkRDIInfoHandler(ARMul_State *state,
                armul_UnkRDIInfoUpcall *proc, void *handle)
```

Remove the upcalls using:

```
int ARMul_RemoveUnkRDIInfoHandler(ARMul_State *state,
                                  void *node)
```

Refer to *Installing an upcall* on page 4-54 for more information.

Example

The `angel.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.

4.10.10 EventUpcall

This upcall catches ARMulator events.

Syntax

```
typedef void armul_EventUpcall(void *handle, unsigned int event,
                               ARMword addr1, ARMword addr2)
```

where:

handle is the handle passed to `ARMul_InstallExitHandler`.

event is one of the event numbers defined in Table 4-3 on page 4-91, Table 4-4 on page 4-92, and Table 4-5 on page 4-92.

addr1 is the first word of the event.

addr2 is the second word of the event.

Usage

Install the upcall using:

```
void *ARMul_InstallEventUpcall(ARMul_State *state,
                               armul_EventUpcall *fn,
                               void *handle)
```

Remove the upcall using:

```
int ARMul_RemoveEventUpcall(ARMul_State *state, void *node)
```


4.11 Memory access functions

The memory model can be probed by another model using a set of functions for reading and writing memory. These functions access memory without inserting cycles on the bus. If your model needs to insert cycles on the bus, it should 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 resulted in a data abort.

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.11.1 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 ARMul_ReadWord(ARMul_State *state, ARMword address)
ARMword ARMul_ReadHalfWord(ARMul_State *state, ARMword address)
ARMword ARMul_ReadByte(ARMul_State *state, ARMword address)
```

where:

state is a pointer to the ARMulator state.

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

Return

The functions return the word, halfword, or byte, as appropriate.

4.11.2 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 ARMul_WriteWord(ARMul_State *state, ARMword address,
                     ARMword data)

void ARMul_WriteHalfWord(ARMul_State *state, ARMword address,
                         ARMword data)

void ARMul_WriteByte(ARMul_State *state, ARMword address,
                     ARMword data)
```

where:

state is a pointer to the ARMulator state.

address is the address in simulated memory to write to.

data is the word, halfword, or byte to write.

4.12 Event scheduling functions

The event scheduling functions enable you to schedule a call to a function based on:

- the number of instructions executed (*instruction events*)
- the number of memory system cycles (*cycle events*)
- the number of core cycles (*core cycle events*).

This section describes the event scheduling functions:

Instruction events The following functions enable you to schedule instruction events:

- *armul_Hourglass* on page 4-68
- *ARMul_HourglassSetRate* on page 4-69.

Cycle events The following functions enable you to schedule or remove cycle events:

- *ARMul_ScheduleEvent* on page 4-70
- *ARMul_ScheduleEventCore* on page 4-72
- *ARMul_ScheduleEventCoreCycles* on page 4-74
- *ARMul_ScheduleEventCoreOrMemory* on page 4-75
- *ARMul_RemoveEvent* on page 4-71
- *ARMul_RemoveEventCore* on page 4-73
- *ARMul_RemoveEventCoreOrMemory* on page 4-76.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.12.1 armul_Hourglass

The `armul_Hourglass()` function provides a mechanism for calling a function at every instruction, or at every `n` instructions for a value of `n` that is set by the `ARMul_HourglassSetRate()` function.

Syntax

```
typedef void armul_Hourglass(void *handle, ARMword pc,
                             ARMword instr)
```

where:

handle is the handle passed to `ARMul_InstallHourglass`.

pc is the program counter.

instr is the instruction about to be executed.

Return

handle to pass to `ARMul_RemoveHourglass` and `ARMul_HourglassSetRate`.

Usage

Install the function in the same way as upcalls:

```
void *ARMul_InstallHourglass(ARMul_State *state,
                             armul_Hourglass *fn,
                             void *handle)
```

Remove the function with:

```
int ARMul_RemoveHourglass(ARMul_State *state, void *node)
```

The remove function returns:

- `TRUE`, if the hourglass function is removed successfully
- `FALSE`, if the hourglass function is not removed successfully.

You can use the `ARMul_HourglassSetRate()` function to change the default rate at which `armul_Hourglass()` is called. See *ARMul_HourglassSetRate* on page 4-69 for details. See also *Installing an upcall* on page 4-54.

4.12.2 ARMul_HourglassSetRate

This function sets the rate at which the `armul_HourGlass()` function is called. By default, the `armul_Hourglass()` function is called every instruction.

Syntax

```
unsigned long ARMul_HourglassSetRate(ARMul_State *state,
                                     void *node,
                                     unsigned long rate)
```

where:

state is a pointer to the ARMulator state.

node is the handle returned from `ARMul_InstallHourglass()` when the upcall was installed.

rate defines the rate at which the function should be called. For example, a value of 1 calls the function every instruction. A value of 100 calls it every 100 instructions.

Return

The function returns the old hourglass rate.

4.12.3 ARMul_ScheduleEvent

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 ARMul_ScheduleEvent(ARMul_State *state,
                         unsigned long delay,
                         armul_EventProc *func,
                         void *handle)
```

where:

state is a pointer to the ARMulator state.

delay specifies the number of cycles to delay before the event function is called.

func is a pointer to the event function to call of type:

```
typedef unsigned armul_EventProc(void *handle)
```

handle is the **void** * handle to pass to the event function.

Note

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

4.12.4 ARMul_RemoveEvent

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

Syntax

```
void ARMul_RemoveEvent(ARMul_State *state, unsigned long when,
                       armul_EventProc *func, void *handle)
```

where:

state is a pointer to the ARMulator state.

when is the memory cycle count at which the event function was to be called.

func is a pointer to the event function to call, of type:

```
typedef unsigned armul_EventProc(void *handle)
```

handle is the **void** * handle to pass to the event function.

Note

Use `ARMul_Time()` to determine *when*.

4.12.5 ARMul_ScheduleEventCore

`ARMmul_ScheduleEventCore()` function schedules events using the absolute core cycle count at which the event will occur. It enables a function to be called at a specified point in the future.

Syntax

```
void ARMul_ScheduleEventCore(ARMul_State *state,
                             armul_EventProc *func,
                             void *handle,
                             unsigned long when)
```

where:

state is a pointer to the ARMulator state.

func is a pointer to the event function to call, of type:

```
typedef unsigned armul_EventProc(void *handle)
```

handle is the **void** * handle to pass to the event function.

when the absolute core cycle count at which the event function is to be called.

Note

This function is supported only by ARM9-based models.

Use `ARMul_ReadCycles` to determine *when*.

4.12.6 ARMul_RemoveEventCore

`ARMmul_RemoveEventCore()` removes a previously scheduled core cycle based event.

Syntax

```
void ARMul_RemoveEventCore(ARMul_State *state,
                           unsigned long when,
                           armul_EventProc *func, void *handle)
```

where:

state is a pointer to the ARMulator state.

when is the core cycle count at which the event function was to be called.

func is a pointer to the event function to call, of type:

```
typedef unsigned armul_EventProc(void *handle)
```

handle is the **void** * handle to pass to the event function.

4.12.7 ARMul_ScheduleEventCoreCycles

`ARMmul_ScheduleEventCoreCycles()` schedules events using core cycles. It enables a function to be called at a specified number of core cycles in the future.

Syntax

```
void ARMul_ScheduleEventCoreCycles(ARMul_State *state,
                                   unsigned long coreCycleDelay,
                                   armul_EventProc *func,
                                   void *handle)
```

where:

state is a pointer to the ARMulator state.

coreCycleDelay
specifies the number of core cycles to delay before the event function is called.

func is a pointer to the event function to call, of type:

```
typedef unsigned armul_EventProc(void *handle)
```

handle is the **void** * handle to pass to the event function.

Note

This function is only supported on ARM9-based models.

4.12.8 ARMul_ScheduleEventCoreOrMemory

`ARMmul_ScheduleEventCoreOrMemory()` schedules either memory or core cycle based events. It provides a convenient means of scheduling events regardless of whether they are core cycle or memory cycle based. It enables a function to be called at a specified point in the future.

Syntax

```
void ARMul_ScheduleEventCoreOrMemory(ARMul_State *state,
                                     unsigned long delay,
                                     armul_EventProc *func,
                                     void *handle,
                                     int coreNotMemory)
```

where:

state is a pointer to the ARMulator state.

delay specifies the number of cycles to delay before the event function is called.

func is a pointer to the event function to call, of type:

```
typedef unsigned armul_EventProc(void *handle)
```

handle is the **void** * handle to pass to the event function.

coreNotMemory
controls whether the event is scheduled using core cycles or memory cycles:

- `FALSE` = memory cycles
- `TRUE` = core cycles.

Note

This function is only supported on ARM9-based models.

4.12.9 ARMul_RemoveEventCoreOrMemory

`ARMmul_RemoveEventCoreOrMemory()` removes a previously scheduled core or memory cycle based event.

Syntax

```
void ARMul_RemoveEventCoreOrMemory(ARMul_State *state,
                                   unsigned long when,
                                   armul_EventProc *func,
                                   void *handle,
                                   int coreNotMemory)
```

where:

state is a pointer to the ARMulator state.

when is the core or memory cycle count at which the event function was to be called.

func is a pointer to the event function to call, of type:

```
typedef unsigned armul_EventProc(void *handle)
```

handle is the **void** * handle to pass to the event function.

coreNotMemory

controls whether the event was scheduled using core cycles or memory cycles:

- `FALSE` = memory
- `TRUE` = core.

4.13 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:

- *ARMul_RaiseError*
- *ARMul_Time* on page 4-79
- *ARMul_Properties* on page 4-79
- *ARMul_CondCheckInstr* on page 4-80
- *ARMul_AddCounterDesc* on page 4-81
- *ARMul_AddCounterValue* on page 4-82
- *ARMul_HaltEmulation* on page 4-83
- *ARMul_EndCondition* on page 4-83
- *ARMul_DoProg* on page 4-84
- *ARMul_DoInstr* on page 4-84.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.13.1 ARMul_RaiseError

Errors of type `ARMul_Error` are returned from a number of initialization and installation functions. These errors should be passed through `ARMul_RaiseError()`. This is a printf-like function that formats the error message associated with an `ARMul_Error` error code.

Syntax

```
ARMul_Error ARMul_RaiseError(ARMul_State *state,
                             ARMul_Error errcode, ...)
```

where:

state is a pointer to the ARMulator state.

errcode is the error code for the error message to be formatted.

`...` are printf-style format specifiers of variadic type.

Return

The function returns the error code it was passed, after formatting the error message.

Example

This function is a printf-style variadic function, and the textual form can be a printf-style format string. For example:

```
interf->handle = (model_state *)malloc(sizeof(model_state));
if (interf->handle == NULL)
     return ARMul_RaiseError(state, ARMulErr_OutOfMemory);
```

For example, the `ARMulErr_MemTypeUnhandled` error message, used by memory models to reject an unrecognized interface type, is declared:

```
ERROR(ARMulErr_MemTypeUnhandled,
      "Memory model '%s' incompatible with bus interface.")
```

and called:

```
return ARMul_RaiseError(state,
                        ARMulErr_MemTypeUnhandled,
                        ModelName);
```

In this case, the debugger displays an error message such as:

```
Memory model 'Flat' incompatible with bus interface.
```

Extending the error file

The file `errors.h` can be extended by adding more errors. However, new errors must be added only at the *end* of the file. Entries are of the form:

```
ERROR(ARMulErr_OutOfMemory, "Out of memory.")
```

This declares an error message, `ARMulErr_OutOfMemory`, with the textual form:

```
    "Out of memory."
```


4.13.2 ARMul_Time

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

Syntax

```
unsigned long ARMul_Time(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

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

4.13.3 ARMul_Properties

This function returns the properties word associated with the processor being simulated.

This is a bitfield of properties, defined in `armdefs.h`.

Syntax

```
ARMword ARMul_Properties(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the properties word. This is a bitfield of properties, defined in `armdefs.h`.

Example

```
if ((ARMul_Properties(state) & ARMul_HasMMU_Prop != 0)
{
    /* processor has an MMU */
    ...
}
```

4.13.4 ARMul_CondCheckInstr

Given an instruction, the `ARMul_CondCheckInstr()` function returns `TRUE` if it would execute given the current state of the PSR flags.

Syntax

```
unsigned ARMul_CondCheckInstr(ARMul_State *state, ARMword instr)
```

where:

state is a pointer to the ARMulator state.

instr is the instruction opcode to check.

Return

The function returns:

- `TRUE` if the instruction would execute
- `FALSE` if the instruction would not execute.

4.13.5 ARMul_AddToSwitch

A peripheral model must call `ARMul_AddToSwitch` to instruct the switch address decoder to instantiate and install the peripheral.

When the switch model is initialized, it calls the memory initialization function of the peripheral.

Syntax

```
ARMul_Error ARMul_AddToSwitch(ARMul_State *state,
                              tag_t model, toolconf config)
```

where:

`state` is the ARMulator state pointer.

`model` is the name of the peripheral model to install.

`config` is the toolconf database for the peripheral model.

`config` must contain an entry for either `Range` or `Mask`, depending on the type of address decoding used by `switch`, see *Switch* on page 4-105.

4.13.6 ARMul_AddCounterDesc

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

Syntax

```
int ARMul_AddCounterDesc(ARMul_State *state,
                         ARMword *arg1,
                         ARMword *arg2,
                         const char *name)
```

where:

state is a pointer to the ARMulator state.

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 *Upcalls* on page 4-53) 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`.

See the implementation in `armflat.c` for an example.

4.13.7 ARMul_AddCounterValue

This function is called when the debugger requests the current statistics values.

Syntax

```
int ARMul_AddCounterValue(ARMul_State *state,
                          ARMword *arg1,
                          ARMword *arg2,
                          bool is64,
                          const ARMword *counter)
```

where:

state is a pointer to the ARMulator state.

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 the current value of the counter.

Return

The function must always return `RDIError_UnimplementedMessage`.

Usage

When ARMulator receives an `RDICycles()` call from the debugger, it uses the `UnkRDIInfoUpcall()` to ask each module in turn to provide the counter values. Each module responds by calling `ARMul_AddCounterValue()`.

Note

It is essential that a module that calls `ARMul_AddCounterDesc()` when `RDIRequestCycleDesc()` is called also calls `ARMul_AddCounterValue()` when `RDICycles()` is called. It must also call both functions the same number of times and in the same order.

4.13.8 ARMul_HaltEmulation

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

Syntax

```
void ARMul_HaltEmulation(ARMul_State *state,
                         unsigned end_condition)
```

where:

state is a pointer to the ARMulator state.

end_condition
is one of the RDIError error values defined in rdi_err.h. Not all of these errors are valid. The debugger interprets *end_condition* and issues a suitable message.

4.13.9 ARMul_EndCondition

This function returns the *end_condition* passed to ARMul_HaltEmulation().

Syntax

```
unsigned ARMul_EndCondition(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The end condition passed to ARMul_HaltEmulation().

4.13.10 ARMul_DoProg

This function starts running the simulator at the current pc value. It is called from the ARMulator RDI interface.

Syntax

```
ARMword ARMul_DoProg(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the value of pc on halting simulation.

4.13.11 ARMul_DoInstr

This function executes a single instruction. It is called from the ARMulator RDI interface.

Syntax

```
ARMword ARMul_DoInstr(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns the value of pc on halting simulation.

4.14 Accessing the debugger

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

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

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:

- *ARMul_DebugPrint*
- *ARMul_ConsolePrint* on page 4-86
- *ARMul_PrettyPrint* on page 4-86
- *ARMul_DebugPause* on page 4-87.

The RDI functions are:

- *ARMul_RDILog* on page 4-87
- *ARMul_HostIf* on page 4-88.

4.14.1 ARMul_DebugPrint

This function displays a message in the RDI logging window under AXD, ADW, or ADU, or to the console under `armsd`.

Syntax

```
void ARMul_DebugPrint(ARMul_State *state, const char *format,
                      ...)
```

where:

state is a pointer to the ARMulator state.

format is a printf-style formatted output string.

... are a variable number of parameters associated with *format*.

4.14.2 ARMul_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 ARMul_ConsolePrint(ARMul_State *state, const char *format,
                        ...)
```

where:

state is a pointer to the ARMulator state.

format is a printf-style formatted output string.

... are a variable number of parameters associated with *format*.

Note

Use `ARMul_PrettyPrint()` to display startup messages.

4.14.3 ARMul_PrettyPrint

This function prints a string in the same way as `ARMul_ConsolePrint()`, but in addition performs line-break checks so that wordwrap is avoided. It should be used for displaying startup messages.

Syntax

```
void ARMul_PrettyPrint(ARMul_State *state, const char *format,
                       ...)
```

where:

state is a pointer to the ARMulator state.

format is a printf-style formatted output string.

... are a variable number of parameters associated with *format*.

4.14.4 ARMul_DebugPause

This function waits for the user to press any key.

Syntax

```
void ARMul_DebugPause(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

4.14.5 ARMul_RDILog

This function returns the value of the RDI logging level.

Syntax

```
ARMword ARMul_RDILog(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

4.14.6 ARMul_HostIf

This function returns a pointer to a `RDI_HostosInterface` structure, defined in `rdi_hif.h`. The structure includes pointers to RDI functions that enable a debug target to send and receive textual information to and from a host.

Syntax

```
const RDI_HostosInterface *ARMul_HostIf(ARMul_State *state)
```

where:

state is a pointer to the ARMulator state.

Return

The function returns a pointer to the `RDI_HostosInterface` structure. Refer to `rdi_hif.h` for the `RDI_HostosInterface` structure definition.

Usage

An operating system model can make use of this to:

- efficiently access the console window (under AXD, ADW, or ADU) or the console (under `armsd`) without going through `ARMul_ConsolePrint()`
- receive user input.

The following input/output functions are included in `RDI_HostosInterface`:

`void writec(RDI_Hif_HostosArg *arg, int c)`
: writes a single character to the console window under AXD, ADW, or ADU, or to the console under `armsd`. This is used by `ARMul_ConsolePrint()`, and by the simulation of `SYS_WriteC` in `angel.c`.

`int readc(RDI_Hif_HostosArg *arg, char const *buffer, int len)`
: reads a single character of input from the host debugger.

`int write(RDI_Hif_HostosArg *arg, char const *buffer, int len)`
: writes a stream of data to the console window under AXD, ADW, or ADU, or to the console under `armsd`.

`char *gets(RDI_Hif_HostosArg *arg, char *buffer, int len)`
: reads a string from the host debugger.

4.15 Tracer

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

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.15.1 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.

4.15.2 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.

4.15.3 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.

4.15.4 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.

4.16 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.

Note

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

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 4-4 on page 4-92
- events from the MMU and cache (not on StrongARM-110), listed in Table 4-3 below
- events from the prefetch unit (ARM8-based processors only), listed in Table 4-5 on page 4-92.

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.

You can catch events by installing an event upcall (see *EventUpcall* on page 4-64). You can raise an event by calling `ARMul_RaiseEvent()` (see *ARMul_RaiseEvent* on page 4-93).

Table 4-3 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

Table 4-4 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 4-5 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

4.16.1 ARMul_RaiseEvent

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

Syntax

```
void ARMul_RaiseEvent(ARMul_State *state, unsigned int event,
                      ARMword word1, ARMword word2)
```

where:

state is a pointer to the ARMulator state.

event is one of the event numbers defined in Table 4-3 on page 4-91, and Table 4-4 on page 4-92 and Table 4-5 on page 4-92.

word1 is the first word of the event (see the Tables above).

word2 is the second word of the event (see the Tables above).

4.17 Map files

The type and speed of memory in a simulated system is 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 `armcul.cnf` (see *armul.cnf, the ARMulator configuration file* on page 4-98). A `.map` file must define rw areas that are at least as large as those specified for the heap and stack in `armul.cnf`, and at the same locations. If this is not the case, Data Aborts are likely to occur during execution.

This section does not apply to ARM10 systems (see *Basic ARM ten system configuration trace files* on page 4-114).

4.17.1 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 may 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 (*) may 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 should 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 may need 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 labelled 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 would be 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 should access, 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 4-2

```
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`.

4.18 armul.cnf, the ARMulator configuration file

`armul.cnf` is a ToolConf configuration file. See *ToolConf* on page 4-108.

Depending on your system, `armul.cnf` may be located in:

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

If you are using armsd on a UNIX system, you can have a local copy of `armul.cnf` in your current working directory. If it finds a copy of `armul.cnf` in your current working directory, ARMulator uses it in preference to the copy in the above location.

`armul.cnf` has the following regions:

- *armul.cnf header* on page 4-100
- *Processors* on page 4-101
- *Memories* on page 4-104
- *Coprocessors* on page 4-106
- *Early models* on page 4-106
- *Late models* on page 4-107.

This is the order of these regions in `armul.cnf` as supplied. The order is not important.

`armul.cnf` is not used by BATS (see *Basic ARM ten system configuration trace files* on page 4-114).

4.18.1 Predefined tags

Before reading armul.cnf, ARMulator creates several tags itself, based on the settings you give to the debugger. These are given in Table 4-6. Preprocessing directives in armul.cnf use these tags to control the configuration.

Table 4-6 Tags predefined by ARMulator

Tag	Description
`RDI_*`	A tag starting `RDI_` is created for each simulator in the DLL. For example, `RDI_BASIC` (the ARM6, ARM7, ARM8 simulator) and `RDI_STRONG` (the StrongARM simulator).
`MEMORY_*`	A tag starting `MEMORY_` is created for each memory model declared in `models.h`. For example, `MEMORY_Flat`.
`COPROCESSOR_*`	A tag starting `COPROCESSOR_` is created for each coprocessor model declared in `models.h`. For example, `COPROCESSOR_Validate`.
`OSMODEL_*`	A tag starting `OSMODEL_` is created for each operating system model declared in `models.h`. For example, `OSMODEL_Angel`.
`MODEL_*`	A tag starting `MODEL_` is created for each basic model. For example, `MODEL_Profiler`.
`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.
`MemConfigToLoad`	Set to a `.map` filename, if one is specified from the debugger.
`UseMapFile`	Set to `True` is a mapfile is to be loaded, `False` if not.

4.18.2 armul.cnf header

The header is at the root level in `armul.cnf`. It contains flags used by preprocessing directives in the file. This allows changes in configuration which affect several locations in the file to be controlled from a single location.

All the flags are Boolean (see *Boolean flags in a ToolConf database* on page 4-111). They are listed in Table 4-7.

Table 4-7 Flags defined in the armul.cnf header

Flag	Description
`Verbose`	Models check the `Verbose` flag to decide whether to report their full configuration during initialization. If you write your own models you should also check this, to ease debugging.
`TraceMemory`	The `TraceMemory` flag controls whether memory accesses are traced by the Tracer model.
`Validate`	`Validate` controls whether ARMulator initializes an ARM validation system. An ARM validation system is a memory or coprocessor model which can generate exceptions and interrupts. It is used to validate some aspects of the ARMulator model.
`WatchPointsEnabled`	The ARMulator can support watchpoints, but it runs much more slowly when it is doing so. The `WatchPointsEnabled` flag allows you to make this choice. Set it to `False` when benchmarking, and `True` when debugging.
`UsePageTables`	The PageTables model writes pagetables to memory, and initializes the cache and MMU on cached processors. You can enable or disable this using the `UsePageTables` flag.
`TimerEnabled`	`TimerEnabled` controls whether the timer model is available. See *Timer* on page 2-27.
`IntCEnabled`	`IntCEnabled` controls whether the interrupt controller model is available. See *Interrupt controller* on page 2-27.
`WDogEnabled`	`WDogEnabled` controls whether the watchdog model is available. See *Watchdog* on page 2-28.

4.18.3 Processors

The processors region is a child ToolConf database (see *ToolConf* on page 4-108). 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.

`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

```
{ Processors

{ ARM7TDM
Processor=ARM7TDM
Core=ARM7
ARMulator=BASIC
Architecture=4T
ARM7TDMI:Processor=ARM7TDMI
}

ARM7TDMI=ARM7TDM
}
```

This declares two processors, ARM7TDM and ARM7TDMI.

`ARM7TDM` has a child of its own. Inside this child are the options for an `ARM7TDM`. For example, its name (`Processor`) is `ARM7TDM`, and the `Core` it uses is `ARM7`. It also contains the entry `ARM7TDMI:Processor=ARM7TDMI`. This declares a child of `ARM7TDM` called `ARM7TDMI`. This child inherits all the features of `ARM7TDM` except the processor name.

After the child, `ARM7TDMI=ARM7TDM` declares the `ARM7TDMI` processor. It specifies that it is based on the `ARM7TDM`.

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.
4. Otherwise, if the tag has a value:
   a. Look up the value in the current region.
   b. If the tag has a child, set the current region to be that child, and return to step 2.
5. Otherwise the configuration is not found, and an error is reported.

For the example `ARM7TDMI`:

1. Find `Processors`.
2. Look up `ARM7TDMI` in `Processors`. This finds `ARM7TDMI=ARM7TDM`.
3. This tag has no child.
4. The tag has value `ARM7TDM` so:
   a. Look up `ARM7TDM` in `Processors`. This finds `{ ARM7TDM`.
   b. Return to step 2 with resulting child (from `{` to `}`).
5. (Step 2) Look up `ARM7TDMI` in this child.
6. (Step 3) `ARM7TDMI` has a child, `Processor=ARM7TDMI`. This is the required configuration. All other features of the configuration are inherited from the parent, `ARM7TDM`.

Adding a new 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`, edit the appropriate part of `armul.cnf`:

```
{ ARM7TDM
Processor=ARM7TDM
Core=ARM7
ARMulator=BASIC
Architecture=4T
ARM7TDMI:Processor=ARM7TDMI
ARM7TASIC:Processor=ARM7TASIC
ARM7TASIC:Memory=MyASIC
}

ARM7TDMI=ARM7TDM
ARM7TASIC=ARM7TDM
```

The three lines containing `ARM7TASIC` have been added:

- `ARM7TASIC:Processor=ARM7TASIC`
  This line is added inside the `ARM7TDM` child. The `ARM7TASIC` tag has a child, that declares the new processor.
- `ARM7TASIC:Memory=MyASIC`
  This line is also inside the `ARM7TDM` child. It extends the `ARM7TASIC` child with a declaration of the memory model used by `ARM7TASIC`. (Usually ARMulator uses whatever memory model is specified in the Memories region.)
- `ARM7TASIC=ARM7TDM`
  This line is outside the `ARM7TDM` region, and tells ARMulator to look in `ARM7TDM` for `ARM7TASIC`.

4.18.4 Memories

The memory system inside ARMulator is hierarchical. This allows cache models, bus tracers and so on to be inserted in the simulated system. The hierarchy is controlled by armul.cnf.

Each processor configuration has a `Memory` tag. This specifies the top level of the memory system. The same algorithm is used to find a memory model configuration as is used to find a processor configuration (see *Finding the configuration for a selected processor* on page 4-102).

Example

```
{ Memories

{ MMUlator

{ ARM700
ARM710:NoCoprocessorInterface
ARM710:ChipNumber=0x710
}

ARM710=ARM700

Memory=Default
}

ARM710=MMUulator
}
```

Following the algorithm given on page 4-102:

1. Find `Memories`.
2. Look up `ARM710` in `Memories`.
3. Tag has no child.
4. Tag has value `MMUlator`, so:
   a. Look up `MMulator` in `Memories`.
   b. Return to step 2 with child.
5. (Step 2) Look up `ARM710` in `Memories:MMUlator`.
6. (Step 3) Tag has no child.
7. (Step 4) Tag has value `ARM700` so:
   a. Look up `ARM700` in `Memories:MMUlator`.
   b. Return to step 2 with child.
8. (Step 2) Look up `ARM710` in `Memories:MMUlator:ARM700`.

9. (Step 3) This has a child. This is the required configuration.

So ARM710 is derived from ARM700 (generic cached ARM7). ARM700 in turn is derived from MMUlator (generic cached ARM).

When the MMUlator cache model is initialized it looks for the next level down in the memory hierarchy. It looks up the Memories tag in the ARM710 configuration, and finds Default by searching back up the tree to the MMUlator region.

4.18.5 Switch

The switch module is a veneer between the processor model and the memory models, including peripheral models. It has an entry in the Memories region. This contains the configuration specifying how switch decodes accesses to main memory.

Peripheral models add their own configuration details dynamically using ARMul_AddToSwitch. One of two types of address decoding can be used:

Range — This specifies that the model it to be used if the address lies within the range specified, including both endpoints.

Mask — This specifies a mask and a value. The model is used if the address AND the mask equals the value.

In the example below, the mask and value specified have the same effect as the range specified.

Example

```
#if MEMORY_Switch
{ Switch
;; The switch memory model multiplexes peripheral models with a
;; memory model

;;; Specify Range or Mask to use range or mask decoding.
;; Decode for the RAM (Memory Model)
; Use RAM from 0->0x7fffffff
Range=0x0,0x7fffffff

; Use RAM for (address & 0x80000000) == 0
;Mask=0x80000000,0x0
}
#endif
```

4.18.6 Coprocessors

The `Coprocessors` region contains two types of entry:

- configurations for all coprocessor models that need them, for example:

  ```
  { Coprocessors
  DummyMMU:ChipID=0x12345678
  }
  ```

- a list of assignments of coprocessor numbers to coprocessor models, for example:

  ```
  Coprocessor[15]=DummyMMU
  ```

  which associates the `DummyMMU` coprocessor model with coprocessor number 15.

The ARM supports 16 coprocessors, numbered 0 to 15. Entries outside this range are ignored.

4.18.7 Early models

The EarlyModels region contains configurations for early basic models.

Early models are initialized before memory initialization. Other models are initialized after memory initialization (see *Basic model interface* on page 4-4).

In all other respects early models conform to the description of late models (see *Late models* on page 4-107).

4.18.8 Late models

The `Models` region contains configurations for late basic models. Late basic models attach themselves to call-backs, write things to memory and so on. They do not provide functions to the ARMulator.

ARMulator goes through the list of models declared to it, and initializes all those with entries in the `Models` region.

For example, the profiler module configuration is contained in the `Models` region:

```
{ Models

{ Profiler
Type=Instruction
}

}
```

This tells ARMulator to start the model called Profiler, which initializes itself to profile instructions.

Disabling models

A model is only started if a configuration for it is found in the Models region. You can disable a model by removing its configuration from the region.

For example, the `Pagetables` model is controlled in this way. In the `Header` region, there is an entry:

```
UsePageTables=True
```

and in `Models`:

```
{ Models

#if UsePageTables==True
{ PageTables
MMU=Yes
AlignFaults=No
Cache=Yes
}
#endif

}
```

If you set `UsePageTables` to `False` in the header, the `#if UsePageTables==True` directive fails, no configuration for `PageTables` is found, and the model is not enabled.

4.19 ToolConf

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. We recommend you to take a copy of `armul.cnf` before modifying it. If you use a different name for each copy, you can select which copy to use (see *Configuring ARMulator to use the example* on page 3-15).

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

ToolConf is not used by BATS (see *Basic ARM ten system configuration trace files* on page 4-114).

4.19.1 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.

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 needs to include 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.

4.19.2 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

4.19.3 SI units in a ToolConf database

Some values may 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.

4.19.4 ToolConf_Lookup

This function performs a lookup on a specified tag in the `armul.cnf` database. 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 `armul.cnf` 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 */
```


4.19.5 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"))
```

4.20 Basic ARM ten system configuration trace files

Configuration trace (CTR) files describe the configurations of the systems that the *Basic ARM Ten System* (BATS) can model. They describe which components are used by the system and how they are interconnected. BATS is quite distinct from other ARMulator models.

You must create a new CTR file if you need to model a system different from the models supplied. We recommend that you copy one of the supplied files and edit the copy. CTR files are in `Install_directory\bin`, and have `.ctr` file extensions.

The sections of a configuration file must be in the order given here.

Any line starting with a # symbol is a comment. You can put comment lines anywhere in a CTR file.

You can also append a comment to a line using the # symbol. The comment ends at the end of the line. We recommend that you do not do this often, because most CTR lines are long.

4.20.1 Setting the time units

The first section in a CTR file sets a fundamental time unit. It has a single line.

Syntax

```
TimeUnitsPerNanosecond "n"
```

where:

n is an integer in decimal notation.

Usage

The debugger system clock counts simulated time. CTR files specify the time required for each kind of cycle in multiples of the fundamental time unit. The default value of *n* is 100. This makes the fundamental time unit 10ps.

If you change *n*, it is your responsibility to ensure that times counted by the debugger system clock are meaningful (see *System time and wait states* on page 4-120).

4.20.2 Opening module classes

The second section in a CTR file is the *Open Classes* section. It has a list of the module classes used. You must not edit this section.

4.20.3 Creating instances

The third section in a CTR file is the *Create Instances* section. It has subsections to create each instance of each module.

4.20.4 The processor instance

This is a subsection in the Create Instances section of the CTR file. You can edit the period of the MCLK signal, the addresses of the stack and heap, and whether semihosting is enabled or disabled.

Syntax

```
CreateInstance "armv5" "processor"
SetParameters "processor" 6
+ "MCLK_PERIOD" "INT32" "period"     "armv5"
+ "HEAP_BASE"   "HEX32" "0xhhhhhhhh" "armv5"
+ "HEAP_LIMIT"  "HEX32" "0xhhhhhhhh" "armv5"
+ "STACK_BASE"  "HEX32" "0xhhhhhhhh" "armv5"
+ "STACK_LIMIT" "HEX32" "0xhhhhhhhh" "armv5"
+ "SEMIHOSTING_ENABLED" "BOOLEAN" "bool" "armv5"
```

where:

period is the period of the **MCLK** signal applied to the `armv5` core, in fundamental time units (see *Setting the time units* on page 4-114).

`0x`*hhhhhhhh* are hexadecimal memory addresses.

bool may be either `TRUE` or `FALSE`.

Entries must be separated by at least one whitespace character.

Example

```
CreateInstance "armv5" "ARM10TDMI"
SetParameters "ARM10TDMI" 6
+ "MCLK_PERIOD" "INT32" "5000"       "armv5"
+ "HEAP_BASE"   "HEX32" "0x30000000" "armv5"
+ "HEAP_LIMIT"  "HEX32" "0x70000000" "armv5"
+ "STACK_BASE"  "HEX32" "0x80000000" "armv5"
+ "STACK_LIMIT" "HEX32" "0x70000000" "armv5"
+ "SEMIHOSTING_ENABLED" "BOOLEAN" "TRUE" "armv5"
```

4.20.5 The AMBA instance

This is a subsection in the Create Instances section of the CTR file.

The `amba` switch module models the AMBA bus.

It has an arbiter that selects which bus master should drive the bus at any time. It also has a decoder that decides which memory module, or peripheral module, should receive each transaction (see *The AMBA memory map* on page 4-117 and *The reference peripherals instance* on page 4-119).

The number of waits required by each memory module is configured in the AMBA module instantiation, not in the memory module instantiations. This is because it is AMBA that applies the waits (see *System time and wait states* on page 4-120).

Syntax

```
CreateInstance "amba"  "AMBA1"
SetParameters  "AMBA1" 2
+ "BCLK_PERIOD" "INT32" "period" "amba"
+ "DEFAULT_GRANT_BUS_MASTER" "INT32" "b" "amba"
```

followed by a memory map. This is described in *The AMBA memory map* on page 4-117.

where:

period is the period of the **BCLK** signal applied to the `amba` bus. This must be the same as the **MCLK** signal applied to the processor core.

b is the number of the **BUS_MASTERS** port linked to the default grant bus master. On reset, the arbiter grants control of the bus to the default grant bus master.

Entries must be separated by at least one whitespace character.

Example

```
CreateInstance "amba" "AMBA1"
SetParameters "AMBA1" 2
+ "BCLK_PERIOD" "INT32" "2000" "amba"
+ "DEFAULT_GRANT_BUS_MASTER" "INT32" "0" "amba"
```

4.20.6 The AMBA memory map

The amba module instantiation in the Create Instances section of the CTR file includes a memory map. You can edit this to model the memory in your system.

Syntax

```
SetTable "ARM10T" "MEMORY_MAP" 6 n
```

followed by *n* lines of memory map of the following format:

```
+ "0xhhhhhhhh" "0xhhhhhhhh" "apply_waits" "n" "s" "port"
```

where:

`0x`*`hhhhhhhh`*
: are hexadecimal memory addresses. These are used by the decoder to route transactions to the appropriate ports.

  Note

  If your simulated system has reference peripherals, you must locate them at `0x0a000000` to `0x0affffff`.

`apply_waits`
: is either `TRUE` or `FALSE`. Set *`apply_waits`* `TRUE` for accurate benchmarking, `FALSE` for faster debugging in single processor systems.

`n`
: is the number of **BCLK** cycle waits required for a nonsequential access to the memory linked to this port. No waits are applied if *`apply_waits`* is `FALSE`.

`s`
: is the number of **BCLK** cycle waits required for a sequential access to the memory linked to this port. No waits are applied if *`apply_waits`* is `FALSE`.

`port`
: is the port number. These must run consecutively from 0.

Entries must be separated by at least one whitespace character.

Example

```
SetTable "AMBA1" "MEMORY_MAP" 6 4
+ "0x00000000" "0x07ffffff" "TRUE"  "5" "5" "0"
+ "0x0a000000" "0x0affffff" "FALSE" "0" "0" "1" # Peripherals
+ "0x10000000" "0x7fffffff" "TRUE"  "5" "5" "2"
+ "0x80000000" "0xffffffff" "TRUE"  "5" "5" "3"
```

4.20.7 Memory instances

For each instance of the `virtmem` memory module, there must be a subsection in the *Create Instances* section of the CTR file.

Syntax

```
CreateInstance "virtmem" "instancename"
SetParameters "instancename" 0
```

where:

instancename is the name of this instance of the *virtmem* memory module. It has values like `mem1`, `mem2`, or `mem3` in the supplied CTR files.

Entries must be separated by at least one whitespace character.

4.20.8 The reference peripherals instance

This is a subsection in the Create Instances section of the ARM1020T_PERIP CTR file.

You can alter which interrupts are initially enabled.

Syntax

```
CreateInstance  "mperf" "mperf0"
SetParameters   "mperf0" 2
+ "INITIAL_FIQENABLE" "INT32" "fiq" "mperf"
+ "INITIAL_IRQENABLE" "INT32" "irq" "mperf"
```

where:

fiq is an integer. When BATS is initialized, the value of *fiq* is written to the FIQEnable register, see *Interrupt controller* on page 4-121.

irq is an integer. When BATS is initialized, the value of *irq* is written to the IRQEnable register, see *Interrupt controller* on page 4-121.

Entries must be separated by at least one whitespace character.

The parameter entries are optional, see examples below. The default values are those shown in the first example. The effect of these values is that inputs FIQ0 and IRQ6 are enabled. IRQ6 is the lowest-numbered external interrupt, see *Interrupt controller defined bits* on page 4-122.

Examples

```
CreateInstance  "mperf" "mperf0"
SetParameters   "mperf0" 2
+ "INITIAL_FIQENABLE" "INT32" "1" "mperf"
+ "INITIAL_IRQENABLE" "INT32" "64" "mperf"

CreateInstance  "mperf" "mperf0"
SetParameters   "mperf0" 1
+ "INITIAL_IRQENABLE" "INT32" "64" "mperf"

CreateInstance  "mperf" "mperf0"
SetParameters   "mperf0" 0
```

4.20.9 Connecting module instances

The fourth section in a CTR file is the *Connect Module Instances* section. It has a subsection for each instance of each module. The subsection specifies the links connected to the instance. You must not edit this section.

4.20.10 Closing instantiations

The final section in a CTR file is the *Finalize Instantiations* section. It closes all the instantiations. You must not edit this section.

4.20.11 System time and wait states

The default fundamental time unit is 10ps (see *Setting the time units* on page 4-114).

MCLK and **BCLK** periods are defined as multiples of the fundamental time unit. See:

- *The processor instance* on page 4-115
- *The AMBA instance* on page 4-116.

Memory waits are defined as multiples of the **BLCK** period. See *The AMBA memory map* on page 4-117.

Only the system clock is affected by changes in fundamental time unit or wait state configuration. This means that these configuration details are important for benchmarking but not for debugging.

Note

MCLK is not gated when a processor core is busy waited by coprocessor 15, and the processor core module counts the busy waited cycles.

4.21 Reference peripherals

Two reference peripherals are detailed here:

- *Interrupt controller*, below
- *Timer* on page 4-123.

BATS has no other reference peripherals.

4.21.1 Interrupt controller

The base address of the interrupt controller, IntBase, is:

- `0x0a000000` in BATS
- configurable in other ARMulator models, see *Interrupt controller* on page 2-27.

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 may be configured to use different bits in the IRQ controller registers, see *Timer* on page 2-27.

This does not apply to BATS, where they must use bits 4 and 5 as shown in Table 4-11.

4.21.2 Timer

The base address of the timer, `TimerBase`, is:

- `0x0A800000` in BATS
- configurable in other ARMulator models, see *Timer* on page 2-27.

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.

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
Angel

This chapter describes the Angel debug monitor. It contains the following sections:

5.1 About 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.

You can use Angel to:

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

These activities require you to have some understanding of how Angel components work together. The more technically challenging ones, such as porting operating systems, require you to modify Angel itself.

A typical Angel system has two main components that communicate through a physical link, such as a serial cable:

Debugger The debugger runs on the host computer. It gives instructions to Angel and displays the results obtained from it. All ARM debuggers support Angel, and you can use any other debugging tool that supports the communications protocol used by Angel.

Angel Debug Monitor
The Angel debug monitor runs alongside the application being debugged on the target platform.

See Figure 5-1 on page 5-6 for an overview of a typical Angel system. The debugger on the host machine sends requests to Angel on the target system. Angel interprets those requests and performs an operation such as inserting an undefined instruction where a breakpoint is required, or reading a portion of memory and sending back a response to the host.

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. Refer to *Angel Debug Protocol* for more information on ADP.

Angel is supplied as:

- a standalone form that is built into the Flash and/or ROM of ARM evaluation and development boards and other, third party boards
- prebuilt images that you can program into ROM or download to Flash
- source files that allow new ports to be built.

ANSI C and C++ libraries that support Angel are supplied with the ADS.

5.1.1 Angel system features

Angel provides the following functionality:

- *Debug support* on page 5-3
- *C library semihosting support* on page 5-3
- *Communications support* on page 5-4
- *Task management* on page 5-5
- *Exception handling* on page 5-5.

See Figure 5-1 on page 5-6 for an overview of the Angel components that provide this functionality.

Debug support

Angel provides the following basic debug support:

- reporting and modifying memory and processor status
- downloading applications to the target system
- setting breakpoints.

Refer to *Angel debugger functions* on page 5-25 for more information on how Angel performs these functions.

C library semihosting support

Angel uses a *software interrupt* (SWI) mechanism to enable applications linked with the ARM C and C++ libraries to make *semihosting* requests. Semihosting requests are requests such as *open a file on the host*, or *get the debugger command line*, that must be communicated to the host to be carried out. These requests are referred to as semihosting because they rely on the C library of the host machine to carry out the request.

The ADS provides prebuilt ANSI C libraries that you can link with your application. Specific C library functions, such as input/output, use the SWI mechanism to pass the request to the host system.

These libraries are used by default when you link code that calls ANSI C library functions. Refer to the description of the C libraries in the *ADS Tools Guide* for more information.

Angel uses a single SWI to request semihosting operations. By default, the SWI is `0x123456` in ARM state and `0xab` in Thumb state. You can change this number if required. Refer to *Configuring Angel* on page 5-37 for more information.

If semihosting support is not required you can disable it by setting the `semihosting_enabled` variable in the ARM debuggers:

- In armsd set:

  ```
  $semihosting_enabled = 0
  ```

- In AXD, ADW or ADU, select **Debugger Internals** from the **View** menu to view and edit the variable. Refer to the description of ARM debuggers in the *ADS Debuggers Guide* for more information.

Refer to Chapter 6 *Semihosting SWIs* for details of the semihosting SWIs.

Communications support

Angel communicates using ADP, and uses *channels* to allow multiple independent sets of messages to share a single communications link. Angel provides an error-correcting communications protocol over:

- Serial and serial/parallel connection from host to the target board, with Angel resident on the board.
- Ethernet connection from the host to ARM development board, with Angel resident on the board. For the ARM development board, this requires the Ethernet Adaptor Kit (No. KPI 0014D), available separately from ARM Limited.

The host and target system channel managers ensure that logical channels are multiplexed reliably. The device drivers detect and reject corrupted data packets. The channel managers monitor the overall flow of data and store transmitted data in buffers, in case retransmission is required. Refer to *Angel communications architecture* on page 5-41 for more information.

The Angel Device Driver Architecture uses Angel task management functionality to control packet processing and to ensure that interrupts are not disabled for long periods of time.

You can write device drivers to use alternative devices for debug communication, such as a ROMulator. You can extend Angel to support different peripherals, or your application can address devices directly.

Task management

All Angel operations, including communications and debug operations, are controlled by Angel task management. Angel task management:

- ensures that only a single operation is carried out at any time
- assigns task priorities and schedules task accordingly
- controls the Angel environment processor mode.

Refer to *Angel task management* on page 5-28 for more information.

Exception handling

Angel exception handling provides the basis for debug, C library semihosting, communications and task management. Angel installs exception handlers for each ARM exception type except Reset.

SWI Angel installs a SWI exception handler to support C library semihosting requests, and to allow applications and Angel to enter Supervisor mode.

Undefined Angel uses three Undefined Instructions to set breakpoints in code. Refer to *Setting breakpoints* on page 5-21 for more information.

Data, Prefetch Abort
Angel installs basic Data and Prefetch Abort handlers. These handlers report the exception to the debugger, suspend the application, and pass control back to the debugger.

FIQ, IRQ Angel installs IRQ and FIQ handlers that enable Angel communications to run off either, or both types of interrupt. If you have a choice you should use IRQ for Angel communications, and FIQ for your own interrupt requirements.

You can chain your own exception handlers for your own purposes. Refer to *Chaining exception handlers* on page 5-19 for more information.

5.1.2 Angel component overview

The main components of an Angel system are shown in Figure 5-1.

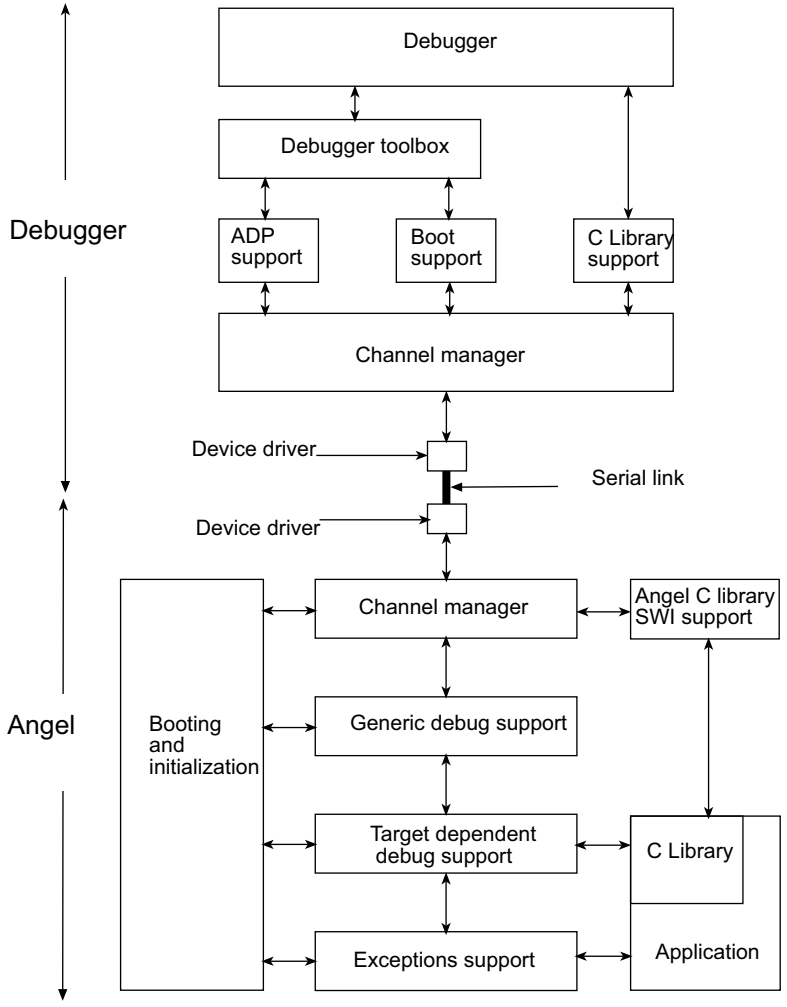

Figure 5-1 A typical Angel system

The following sections give a summary of the system components:

Host system components summary

The host system components are:

Debugger This is the *ARM Debugger for Windows* (ADW or AXD), the *ARM Debugger for UNIX* (ADU), the ARM command-line debugger (armsd), or a third party debugger that supports the Angel Debug Protocol.

Debugger toolbox

This provides an interface between the debugger and the *Remote Debug Interface* (RDI).

ADP support

This translates between RDI calls from the debug controller and Angel ADP messages.

Boot support

This establishes communication between the target and host systems. For example, it sets baud rates and re-initializes Angel in the target.

C library support

This handles semihosting requests from the target C library.

Host channel manager

This handles the communication channels, providing the functionality of the devices at a higher level.

Device drivers

These implement specific communications devices on the host. Each driver provides the entry points required by the channel manager.

Target system components summary

The target system components are:

Device drivers

These implement specific communications devices on the ARM development boards. Each driver provides the entry points required by the channel manager.

Channel manager

This handles the communication channels. It provides a streamed packet interface that hides details of the device driver in use.

Generic debug support

This handles the ADP by communicating with the host over a configured channel, and sending and receiving commands from the host.

Target-dependent debug support

This provides system-dependent features, such as setting up breakpoints and reading/writing memory.

Exceptions support

This handles all ARM exceptions.

C library support

C library support consists of the ARM ANSI C libraries supplied with ADS, and the semihosting support that is built into Angel to send requests to the host when necessary.

Booting and initialization

The Angel booting and initialization support code:

- performs startup checks
- sets up memory, stacks, and devices
- sends a boot message to the debugger.

User application

This is an application on the target system.

5.1.3 Angel system resource requirements

Where possible, Angel resource usage can be statically configured at compile and link time. For example, the memory map, exception handlers, and interrupt priorities are all fixed at compile and link time. Refer to *Configuring Angel* on page 5-37 for more information.

System resources

Angel requires the following non-configurable resources:

- two ARM Undefined Instructions (for big endian or little endian versions)
- one Thumb Undefined Instruction.
- one ARM SWI at `0x123456`
- one Thumb SWI at `0xAB`.

ROM and RAM requirements

Angel requires ROM or Flash memory to store the debug monitor code, and RAM to store data. The amount of ROM, Flash, and RAM required varies depending on the development board you are using.

Additional RAM might be required to download a new version of Angel.

Exception vectors

Angel requires some control over the ARM exception vectors. Exception vectors are initialized by Angel, and are not written to after initialization. This supports systems with ROM at address 0, where the vectors cannot be overwritten.

Note

An application that chains the vectors must unchain them on exit, or the target must be reset, so that the exceptions do not crash the machine when the application is overwritten.

Angel installs itself by initializing the vector table so that it takes control of the target when an exception occurs. For example, debug communications from the host cause an interrupt that halts the application and calls the appropriate code within Angel.

Interrupts

Angel requires use of at least one interrupt to support communication between the host and target systems. You can set up Angel to use:

- IRQ
- FIQ
- both IRQ and FIQ.

It is recommended that you use FIQ for your own interrupt requirements because Angel has no fast interrupt requirements.

Stacks

Angel requires control over its own Supervisor stack. If you want to make Angel calls from your application you *must* set up your own stacks. Refer to *Developing an application under Angel* on page 5-16 for more information.

Angel also requires that the current stack pointer points to a few words of usable full descending stack whenever an exception is possible, because the Angel exception return code uses the application stack to return.

5.2 Developing applications with Angel

This section describes how you can develop applications under Angel.

It also describes the programming restrictions that you must be aware of when developing an application under Angel, and provides some workarounds for Angel intrusions.

Angel is a standalone system that resides on the target board and is always active. Angel is used during the development of the application code. It supports all debugger functions and you can use it to:

- download your application image from a host
- debug your application code
- develop the application before converting to standalone code.

Angel is supplied in the following forms:

In target board ROM

The ARM development and evaluation boards are supplied with Angel built into ROM, or Flash, or both. To use Angel in this form you simply connect your target board to a host machine running a debugger, such as AXD, ADW, ADU, or armsd.

Prebuilt images

Angel is supplied as prebuilt images for the ARM development board board with ADS. These are located in:

- *Install_directory*`\Angel\Images\pid\little` for a little-endian configuration of the ARM development board
- *Install_directory*`\Angel\Images\pid\big` for a big-endian configuration of the ARM development board.

The supplied binaries are:

`angel.rom` This is a ROM image of Angel. You can use this image in place of the Angel in your target board ROM if your board contains an older version. In addition, if you are porting Angel to your own hardware this image provides you with a working default to test against.

`angel.hex` This is an Intellec Hex format version of the Angel ROM.

`angel.m32` This is a Motorola M32 version of Angel.

`angel.elf` This is the ELF image used to build the binary images.

Source code

You can port the Angel source code to your own development board if you are developing an application on your own hardware.

5.2.1 Overview of the development procedure

This section gives an overview of the development process of an application using Angel, from the evaluation stage to the final product.

The stages in the Angel development procedure are:

1. *Stage 1: Evaluating applications* on page 5-13
2. *Stage 2: Building applications on a development board with high dependence on Angel* on page 5-14
3. *Stage 3: Building applications on a development board with little dependence on Angel* on page 5-15
4. *Stage 4: Moving an application to final production hardware* on page 5-16.

Figure 5-2 shows an example of this development procedure. The stages of the development procedure are described in more detail below.

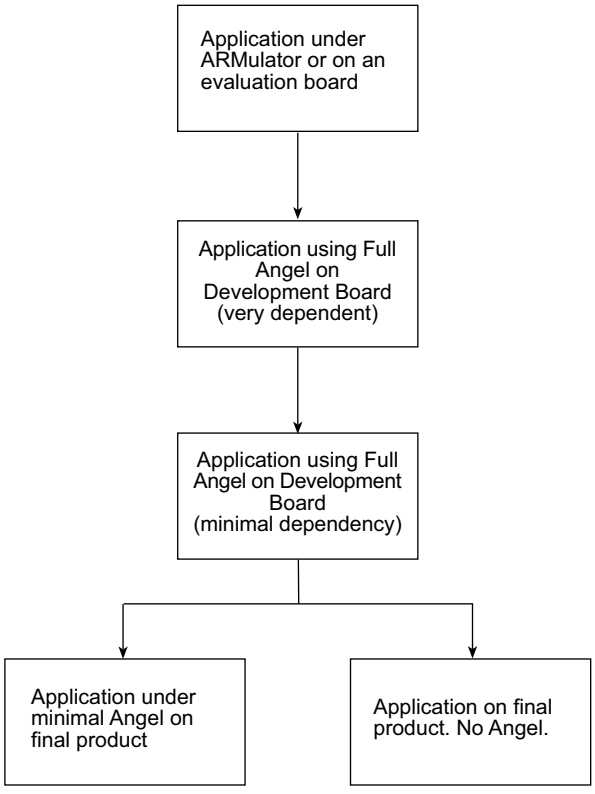

Figure 5-2 The Angel development process

Stage 1: Evaluating applications

If you want to evaluate the ARM processor you must have a program, or suite of programs to run on the ARM processor.

You can rebuild your programs using the ADS, and link them with an ARM C or C++ library.

You can run your ported applications in two ways:

ARMulator You can run your programs under the ARMulator, and evaluate cycle counts to see if the performance is sufficient.

This method does not involve Angel. However, you can use ARM C or C++ library functions that require semihosting because the ARMulator supports the semihosting SWIs. C library calls, unless you have created new implementations of them, are handled by the host C library support.

Evaluation board

Instead of testing programs under the ARMulator, you can use an ARM evaluation board to evaluate performance. In this case you use Angel running as a debug monitor on the ARM evaluation board. You do not have to rebuild Angel, or to be familiar with the way Angel works.

You can build images that are linked with an ARM C or C++ library, and then download the images with an ARM debugger.

Stage 2: Building applications on a development board with high dependence on Angel

After evaluating your application you move to the development stage. At this stage, the target board is either your own development board or an ARM development board:

Using an ARM development board

You can use an ARM development board to closely emulate the configuration of your production hardware. You can develop your application on the board and port it to your final hardware with minimal effort.

Using your own development board

If you are developing on your own hardware it is likely to have different peripheral hardware, different memory maps, and so on from the ARM evaluation boards or development boards. This means that you must port Angel to your development board. The porting procedure includes writing device drivers for (at least one of) your hardware devices.

When you have chosen your development platform, you build a standalone application that runs next to Angel on your target hardware. The procedure for downloading the application to the development platform will depend on the development board you are using. Typical download procedures are described in *Downloading new application versions* on page 5-23.

At this stage you are highly reliant on Angel to debug your application. In addition you must make design decisions about the final form of your application. In particular you should decide whether the final application is standalone, or uses a customized version of Angel to provide initialization code, interrupt handlers, and device drivers.

If you are developing simple embedded applications, you might want to move straight to building your application on a development board.

Stage 3: Building applications on a development board with little dependence on Angel

As you proceed with your development project and your code becomes more stable, you will rely less on Angel for debugging and support. For example, you might want to use your own initialization code, and you might not require C library semihosting support:

- You can switch off semihosting, without building a special cut-down version of Angel, by setting the `$semihosting_enabled` variable in the ARM debuggers. In armsd:

  ```
  semihosting_enabled = 0
  ```

  In ADW or ADU select **Debugger Internals** from the **View** menu to view and edit the variable. Refer to the descriptions of the ARM debuggers in the *ADS Debuggers Guide* for more information.

- You can build an application that links with a customized version of the Angel library. This can be blown into a ROM, soft-loaded into Flash by the ARM debuggers, or installed using a ROM emulator or Multi-ICE.

  If you want to debug a customized version of an Angel application and your hardware supports JTAG you can use Multi-ICE. This requires very few resources on the target.

- You can build an application that uses redefined I/O functions in place of the semihosted I/O functions in the C library. For example, you can provide I/O functions that work with your hardware and keep the Angel functionality for debugging.

Stage 4: Moving an application to final production hardware

When you are ready to move the application onto final production hardware, you have a different set of requirements. For example:

- Production hardware might not have as many communications links as your development board. You might not be able to communicate with the debugger.
- RAM and ROM might be limited.
- Interrupt handlers for timers might be required in the final product, but debug support code is not.

At this stage you can use all the standard C library if you avoid, or have redefined, the I/O functions that use semihosting.

5.2.2 Developing an application under Angel

This section gives useful information on how to develop applications under Angel:

- *Planning your development project*
- *Programming restrictions* on page 5-17
- *Using Angel with an RTOS* on page 5-18
- *Using Supervisor mode* on page 5-19
- *Chaining exception handlers* on page 5-19
- *Linking Angel C library functions* on page 5-20
- *Using assertions when debugging* on page 5-20
- *Setting breakpoints* on page 5-21
- *Changing from little-endian to big-endian Angel* on page 5-21.

Planning your development project

Before you begin your development project you must make basic decisions about such things as:

- the ATPCS variant to be used for your project
- whether or not ARM/Thumb interworking is required
- the endianness of your target system.

Refer to the appropriate chapters of the *ADS Debuggers Guide* and *ADS Developer Guide* for more information on interworking ARM and Thumb code, and specifying APCS options.

In addition, you should consider:

- Whether or not you require C library support in your final application. You must decide how you will implement C library I/O functions if they are required, because the Angel semihosting SWI mechanism will not be available. Refer to *Linking Angel C library functions* on page 5-20 for more information.
- Whether or not the image is built with debug enabled. You should be aware of the small size overhead when using debuggable images as production code.
- Communications requirements. You must write your own device drivers for your production hardware.
- Memory requirements. You must ensure that your hardware has sufficient memory to hold both Angel and your program images.

Programming restrictions

Angel resource requirements introduce a number of restrictions on application development under Angel:

- Angel requires control of its own Supervisor stack. If you are using an RTOS you must ensure that it does not change processor state while Angel is running. Refer to *Using Angel with an RTOS* on page 5-18 for more information.
- You should avoid using ARM SWI `0x123456` or Thumb SWI `0xab`. These SWIs are used by Angel to support C library semihosting requests. Refer to *Configuring SWI numbers* on page 5-40 for information on changing the default SWI numbers.
- If you are using SWIs in your application, and using Multi-ICE for debugging, you should usually set a breakpoint on the SWI handler routine, where you know it is a SWI, rather than at the SWI vector itself.
- If you are using SWIs in your application you must restore registers to the state that they were when you entered the SWI.
- If you want to use the Undefined Instruction exception for any reason you must remember that Angel uses this to handle breakpoints and the exception must be chained.

Using Angel with an RTOS

From the application perspective Angel is single threaded, modified by the ability to use interrupts provided the interrupt is not context switching. External functions must not change processor modes through interrupts. This means that running Angel and an RTOS together is difficult, and is not recommended unless you are prepared for a significant amount of development effort.

If you are using an RTOS you will have difficulties with contention between the RTOS and Angel when handling interrupts. Angel requires control over its own stacks, task scheduling, and the processor mode when processing an IRQ or FIQ.

An RTOS task scheduler must not perform context switches while Angel is running. Context switches should be disabled until Angel has finished processing.

For example:

1. An RTOS installs an ISR to perform interrupt-driven context switches.
2. The ISR is enabled when Angel is active (for example, handling a debug request).
3. An interrupt occurs when Angel is running code.
4. The ISR switches the Angel context, not the RTOS context.

That is, the ISR puts values in processor registers that relate to the application, not to Angel, and it is very likely that Angel will crash.

There are two ways to avoid this situation:

- Detect ISR calls that occur when Angel is active, and do not task switch. The ISR can run, provided the registers for the other mode are not touched. For example, timers can be updated.
- Disable either IRQ or FIQ interrupts, the one Angel is not using, while Angel is active. This is not easy to do.

In summary, the normal process for handling an IRQ under an RTOS is:

1. IRQ exception generated.
2. Do any urgent processing.
3. Enter the IRQ handler.
4. Process the IRQ and issue an event to the RTOS if required.
5. Exit by way of the RTOS to switch tasks if a higher priority task is ready to run.

Under Angel this procedure must be modified to:

1. IRQ exception generated.
2. Do any urgent processing.
3. Check whether Angel is active:
   a. If Angel is active then the CPU context must be restored on return, so scheduling cannot be performed, although for example a counter could be updated. Exit by restoring the pc to the interrupted address.
   b. If Angel is not active, process as normal, exiting by way of the scheduler if required.

Using Supervisor mode

If you want your application to execute in Supervisor mode at any time, you must set up your own Supervisor stack. If you call a SWI while in Supervisor mode, Angel uses four words of your Supervisor stack when entering the SWI. After entering the SWI Angel uses its own Supervisor stack, not yours.

This means that, if you set up your own Supervisor mode stack and call a SWI, the Supervisor stack pointer register (sp_SVC) must point to four words of a full descending stack in order to provide sufficient stack space for Angel to enter the SWI.

Chaining exception handlers

Angel provides exception handlers for the Undefined, SWI, IRQ/FIQ, Data Abort, and Prefetch Abort exceptions. If you are working with exceptions you must ensure that any exception handler that you add is chained correctly with the Angel exception handlers. Refer to the description of processor exceptions in *ADS Developer Guide* for more information.

If you are chaining an interrupt handler and you know that the next handler in the chain is the Angel interrupt handler, you can use the Angel interrupt table rather than the processor vector table. You do not have to modify the processor vector table. The Angel interrupt table is easier to manipulate because it contains the 32-bit address of the handler. The processor vector table is limited to 24-bit addresses.

Note

If your application chains exception handlers (including ISRs) Angel must be reset with a hardware reset if the application is killed. This ensures that the vectors are set up correctly when the application is restarted.

The consequences of not passing an exception on to Angel from your exception handler depend on the type of exception, as follows:

Undefined You will not be able to single step or set breakpoints from the debugger.

SWI If you do not implement the EnterSVC SWI, Angel will not work. If you do not implement any of the other SWIs you will not be able to use semihosting.

Prefetch Abort
The exception will not be trapped in the debugger.

Data Abort The exception will not be trapped in the debugger. If a Data Abort occurs during a debugger-originated memory read or write, the operation might not proceed correctly, depending on the action of the handler.

IRQ This depends on how Angel is configured. Angel will not work if it is configured to use IRQ as its interrupt source.

FIQ This depends on how Angel is configured. Angel will not work if it is configured to use FIQ as its interrupt source.

Linking Angel C library functions

The C libraries provided with the ADS use SWIs to implement semihosting requests. For more information on using libraries, refer to the *ADS Tools Guide*.

You two options for using ARM C library functionality:

- Use the ARM C library semihosting functions for early prototyping and redefine individual library I/O functions with your own C functions targeted at your hardware and operating system environment.
- Support SWIs in your own application or operating system and use the ARM C libraries as provided.

Using assertions when debugging

To speed up debugging, Angel includes runtime assertion code that checks that the state of Angel is as expected. The Angel code defines the `ASSERT_ENABLED` option to enable and disable assertions.

If you use assertions in your code you should wrap them in the protection of `ASSERT_ENABLED` macros so that you can disable them in the final version if required.

```
#if ASSERT_ENABLED
...
#endif
```

Angel uses such assertions wherever possible. For example, assertions are made when it is assumed that a stack is empty, or that there are no items in a queue. You should use assertions whenever possible when writing device drivers. The `ASSERT` macro is available if the code is a simple condition check (variable = value).

Setting breakpoints

Angel can set breakpoints in RAM only. You cannot set breakpoints in ROM or Flash.

In addition, you must be careful when using single step or breakpoints on the UNDEF, IRQ, FIQ, or SWI vectors. Do not single step or set breakpoints on interrupt service routines on the code path used to enter or exit Angel.

Changing from little-endian to big-endian Angel

Changing memory byte order is dependent on the ARM development board you are using. Refer to the documentation that was supplied with the board.

5.2.3 Application communications

Angel requires use of at least one device driver for its own communications requirements. If you are using Angel on a board with more than one serial port, such as the ARM development board, you can either:

- use Angel on one serial port and your own device on the other
- use a customized version of Angel that requires no serial port and use either or both of the serial ports for your application.

The ARM development board Angel port provides examples of raw serial drivers. Refer to the Angel source code for details of how these are implemented. If you want to use Angel with your own hardware you must write your own device drivers.

Angel serial drivers

Figure 5-3 gives an overview of the Angel serial device architecture.

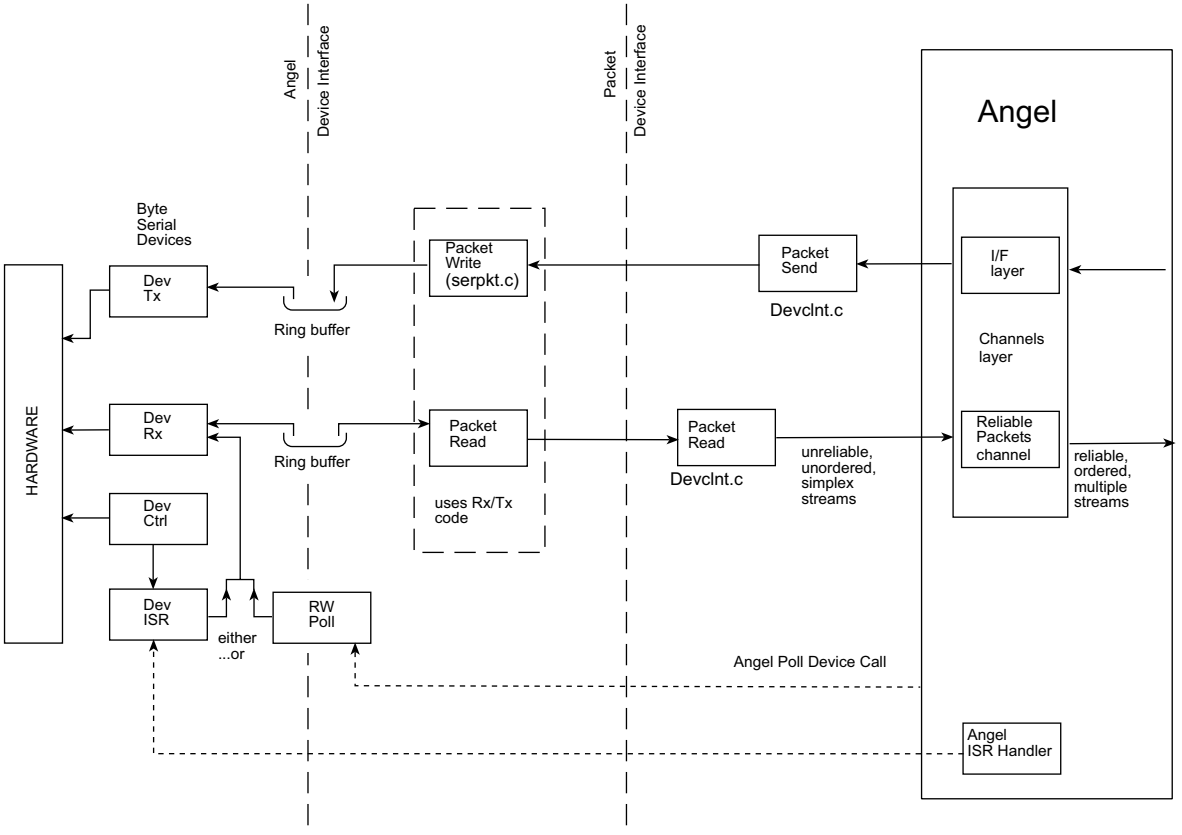

Figure 5-3 Serial device architecture

Using the Debug Communication Channel

You can use `cin` and `cout` in armsd, the channel viewer interface, or the ARM debugger GUI to access the DCC from the host. You can use the DCC channel to send ARM DCC instructions to the processor. No other extra channels are supported.

5.2.4 Downloading new application versions

There are five techniques you can use to move successive versions of your application onto a ARM development board. Each has advantages and disadvantages:

Using Angel with a serial port

This gives slow downloading, but has the advantage that it requires only a simple UART on the ARM development board. If your board supports Flash download you can use this method to fix your image in Flash.

Using Angel with an Ethernet connection

This provides fast downloading, but requires Ethernet hardware on the ARM development board and Ethernet support software to run on the ARM development board. If your board supports Flash download you can use this method to fix your image in Flash.

Flash download

This provides slow to fast downloading, depending on the type of connection you are using.

This method is only available on boards that have Flash memory and are supported by a Flash download program (either the boot Flash monitor on a development board, the ADS Flash download utility, or the ARM Firmware Suite utility). It has the advantage that, after the Flash is set, the image is fixed in memory, even if the board is reset.

You can also download application-only images using this method, but you might not be able to use Angel.

Refer to your ARM development board documentation for more information on downloading to Flash.

Using a ROM emulator to download a new ROM image

Depending on the development board you are using, you might be able to download a ROM image with a ROM emulator. See the documentation that was provided with your board.

If you use one of the ROM replacement methods then you must change from building application images to building ROM images as soon as the development phase starts.

If you use a simple download method then the transition to the development phase is easier because you can move to building ROM images when everything else is working and you are preparing to move to production hardware.

For more information on using the Flash download utility, refer to the *ADS Tools Guide*.

If you are using an EPROM programmer to program big-endian code into 16-bit devices, refer to the fromELF utility information in the *ADS Tools Guide*.

5.3 Angel in operation

This section briefly explains Angel operations you should understand before porting Angel to your own hardware. It contains the following:

- *Initialization*
- *Waiting for debug communications* on page 5-25
- *Angel debugger functions* on page 5-25
- *Angel task management* on page 5-28
- *Context switching* on page 5-32
- *Example of Angel processing: a simple IRQ* on page 5-35.

5.3.1 Initialization

The initialization of the code environment and system is almost identical, whether the code is to initialize the debugger or to launch an application. The initialization sequence is:

1. The processor is switched from the current privileged mode to Supervisor mode with interrupts disabled. Angel checks for the presence of an MMU. If an MMU is present it can be initialized after switching to Supervisor mode.

2. Angel sets the code execution and vector location, depending on the compilation addresses generated by the values of `ROADDR` and `RWADDR`. Refer to *Configuring where Angel runs* on page 5-38 for more information.

3. Code and data segments for Angel are copied to their execution addresses.

4. If the application is to be executed then the runtime system setup code and the application itself are copied to their execution addresses. If the system has ROM at address 0 and the code is to be run from ROM, only the Data and Zero Initialization areas are copied.

5. The stack pointers are set up for each processor mode that Angel operates in. Angel maintains control of its own stacks separately from any application stacks. You can configure the location of Angel stacks. Refer to *Configuring the memory map* on page 5-37 for more information.

6. Target-specific functions such as MMU or Profiling Timer are initialized if they are included in the system.

7. The Angel serializer is set up. Refer to the *Angel task management* on page 5-28 for more information on the Angel serializer.

8. The processor is switched to User mode and program execution is passed to the high level initialization code for the C library and Angel C functions.

   When initialization is complete, program execution is directed to the `__main` entry point.

9. At this point, the initialization sequence is executed:

   a. The communications channels are initialized for ADP.
   b. Any raw data channels installed for the application are set up if you are using extra channels. The application can set this up itself. Refer to the Angel source code for details.
   c. Angel transmits its boot message through the boot task and waits for communication from the debugger.

5.3.2 Waiting for debug communications

After initialization, Angel enters the idle loop and continually calls the device polling function. This ensures that any polled communications device is serviced regularly. When input is detected, it is placed into a buffer and decoded into packet form to determine the operation that has been requested. If an acknowledgment or reply is required, it is constructed in an output buffer ready for transmission.

All Angel operations are controlled by Angel task management. Refer to *Angel task management* on page 5-28 and *Example of Angel processing: a simple IRQ* on page 5-35 for more information on Angel task management.

5.3.3 Angel debugger functions

This section gives a summary of how Angel performs the basic debugger functions:

- reporting memory and processor status
- downloading a program image
- setting breakpoints.

Reporting processor and memory status

Angel reports the contents of memory and the processor registers as follows:

Memory The memory address being examined is passed to a function that copies the memory as a byte stream to the transmit buffer. The data is transmitted to the host as an ADP packet.

Registers Processor registers are saved into a data block when Angel takes control of the target (usually at an exception entry point). When processor status is requested, a subset of the data block is placed in an ADP packet and transmitted to the host.

When Angel receives a request to change the contents of a register, it changes the value in the data block. The data block is stored back to the processor registers when Angel releases control of the target and execution returns to the target application.

Download

When downloading a program image to your board, the debugger sends a sequence of ADP memory write messages to Angel. Angel writes the image to the specified memory location.

Memory write messages are special because they can be longer than other ADP messages. If you are porting Angel to your own hardware your device driver must be able to handle messages that are longer than 256 bytes. The actual length of memory write messages is determined by your Angel port. Message length is defined in `devconf.h` with:

```
#define BUFFERLONGSIZE
```

Setting breakpoints

Angel uses three Undefined Instructions to set breakpoints. The instruction used depends on:

- the endianness of the target system
- the processor state (ARM or Thumb).

ARM state In ARM state, Angel recognizes the following words as breakpoints:

`0xE7FDDEFE` for little-endian systems.

`0xE7FFDEFE` for big-endian systems.

Thumb state In Thumb state, Angel recognizes `0xDEFE` as a breakpoint.

Note

These are not the same as the breakpoint instructions used by Multi-ICE.

These instructions are used for normal, user interrupt, and vector hit breakpoints. In all cases, no arguments are passed in registers. The breakpoint address itself is where the breakpoint occurs.

When you set a breakpoint, Angel:

- stores the original instruction to ensure that it is returned if the area containing it is examined
- replaces the instruction with the appropriate Undefined Instruction.

The original instruction is restored when the breakpoint is removed, or when a request to read the memory that contains the instruction is made in the debugger. When you step through a breakpoint, Angel replaces the saved instruction and executes it.

Note

Angel can set breakpoints only on RAM locations.

When Angel detects an Undefined Instruction it:

1. Examines the instruction by executing an:
   - `LDR` instruction from lr – 4, if in ARM state
   - `LDR` instruction from lr – 2, if in Thumb state.
2. If the instruction is the predefined breakpoint word for the current processor state and endianness, Angel:
   a. halts execution of the application
   b. transmits a message to the host to indicate the breakpoint status
   c. executes a tight poll loop and waits for a reply from the host.

   If the instruction is not the predefined breakpoint word, Angel:
   a. reports it to the debugger as an undefined instruction
   b. executes a tight poll loop and waits for a reply from the host.

ARM breakpoints are detected in Thumb state. When an ARM breakpoint is executed in Thumb state, the Undefined Instruction vector is taken whether executing the instruction in the top or bottom half of the word. In both cases these correspond to a Thumb Undefined Instruction and result in a branch to the Thumb Undefined Instruction handler.

Note

Thumb breakpoints are not detected in ARM state.

5.3.4 Angel task management

All Angel operations are controlled by Angel task management that:

- assigns task priorities and schedules tasks accordingly
- controls the Angel environment processor mode.

Angel task management requires control of the processor mode. This can impose restrictions on using Angel with an RTOS. Refer to *Using Angel with an RTOS* on page 5-18 for more information.

Task priorities

Angel assigns task priorities to tasks under its control. Angel ensures that its tasks have priority over any application task. Angel takes control of the execution environment by installing exception handlers at system initialization. The exception handlers enable Angel to check for commands from the debugger and process application semihosting requests.

Angel will not function correctly if your application or RTOS interferes with the execution of the interrupt, SWI or Data Abort exception handlers. Refer to *Chaining exception handlers* on page 5-19 for more information.

When an exception occurs, Angel either processes it completely as part of the exception handler processing, or calls `Angel_SerialiseTask()` to schedule a task. For example:

- When a SWI occurs, Angel determines whether the SWI is a *simple* SWI that can be processed immediately, such as the EnterSVC SWI, or a *complex* SWI that requires access to the host communication system, and therefore to the serializer. Refer to *Input/Output SWIs* on page 6-10 for more information.
- When an IRQ occurs, the Angel development board port determines whether or not the IRQ signals the receipt of a complete ADP packet. If it does, Angel task management is called to control the packet decode operation. Refer to *Example of Angel processing: a simple IRQ* on page 5-35 for more information. Other Angel ports can make other choices for IRQ processing, provided the relevant task is eventually run.

The task management code maintains two values that relate to priority:

Task type The task type indicates type of task being performed. For example, the application task is of type `TP_Application`, and Angel tasks are usually `TP_AngelCallback`. The task type labels a task for the lifetime of the task.

Task priority The task priority is initially derived from the task type, but thereafter it is independent. Actual priority is indicated in:

- the value of a variable in the task structure
- the relative position of the task structure in the task queue.

The task priority of the application task changes when an application SWI is processed, to ensure correct interleaving of processing.

Table 5-1 shows the relative task priorities used by Angel.

Table 5-1 Task priorities

Priority	Task	Description
Highest	`AngelWantLock`	High priority callback.
-	`AngelCallBack`	Callbacks for Angel.
-	`ApplCallBack`	Callbacks for the user application.
-	`Application`	The user application.
-	`AngelInit`	Boot task. Emits boot message on reset and then exits.
Lowest	`IdleLoop`	Waiting for task

Angel task management is implemented through the following top-level functions:

- `Angel_SerialiseTask()`
- `Angel_NewTask()`
- `Angel_QueueCallback()`
- `Angel_BlockApplication()`
- `Angel_NextTask()`
- `Angel_Yield()`
- `Angel_Wait()`
- `Angel_Signal()`
- `Angel_TaskID()`.

Some of these functions call other Angel functions not documented here. The functions are described in brief below. For full implementation details, refer to the source code in `serlock.h`, `serlock.c`, and `serlasm.s`.

Angel_SerialiseTask

In most cases this function is the entrance function to Angel task management. The only tasks that are not a result of a call to `Angel_SerialiseTask()` are the boot task, the idle task, and the application. These are all created at startup. When an exception occurs, `Angel_SerialiseTask()` cleans up the exception handler context and calls `Angel_NewTask()` to create a new high priority task. It must be entered in a privileged mode.

Angel_NewTask

`Angel_NewTask()` is the core task creation function. It is called by `Angel_SerialiseTask()` to create task contexts.

Angel_QueueCallback

This function:

- queues a packet notification callback task
- specifies the priority of the callback
- specifies up to four arguments to the callback.

The callback executes when all tasks of a higher priority have completed. Table 5-1 on page 5-29 shows relative task priorities.

Angel_BlockApplication

This function is called to allow or disallow execution of the application task. The application task remains queued, but is not executed. If Angel is processing an application SWI when `Angel_BlockApplication()` is called, the block might be delayed until just before the SWI returns.

Angel_NextTask

This is not a function, in that it is not called directly. `Angel_NextTask()` is executed when a task returns from its main function. This is done by setting the link register to point to `Angel_NextTask()` on task entry.

The `Angel_NextTask()` routine:

- enters Supervisor mode
- disables interrupts
- calls `Angel_SelectNextTask()` to select the first task in the task queue that has not been blocked and run it.

Angel_Yield

This is a yield function for polled devices. It can be called:

- by the application
- by Angel while waiting for communications on a polled device
- within processor-bound loops such as the idle loop.

`Angel_Yield()` uses the same serialization mechanism as IRQ interrupts. Like an IRQ, it can be called from either User or Supervisor mode and returns cleanly to either mode. If it is called from User mode it calls the `EnterSVC` SWI to enter Supervisor mode, and then disables interrupts.

Angel_Wait

`Angel_Wait()` works in conjunction with `Angel_Signal()` to enable a task to wait for a predetermined event or events to occur before continuing execution. When `Angel_Wait()` is called, the task is blocked unless the predetermined event has already been signalled with `Angel_Signal()`.

`Angel_Wait()` is called with an event mask. The event mask denotes events that will result in the task continuing execution. If more than one bit is set, any one of the events corresponding to those bits will unblock the task. The task remains blocked until some other task calls `Angel_Signal()` with one or more of the event mask bits set. The meaning of the event mask must be agreed beforehand by the routines.

If `Angel_Wait()` is called with a zero event mask, execution continues normally.

Angel_Signal

`Angel_Signal()` works in conjunction with `Angel_Wait()`. This function sends an event to a task that is now waiting for it, or will in the future wait for it:

- If the task is blocked, `Angel_Signal()` assumes that the task is waiting and subtracts the new signals from the signals the task was waiting for. The task is unblocked if the event corresponds to any of the event bits defined when the task called `Angel_Wait()`.
- If the task is running, `Angel_Signal()` assumes that the task will call `Angel_Wait()` at some time in the future. The signals are marked in the task `signalWaiting` member.

`Angel_Signal()` takes a task ID that identifies a current task, and signals the task that the event has occurred. See the description of `Angel_Wait()` for more information on event bits. The task ID for the calling task is returned by the `Angel_TaskID()` macro. The task must write its task ID to a shared memory location if an external task is to signal it.

Angel_TaskID

This macro returns the task ID (a small integer) of the task that calls it.

Angel_TaskIDof

This macro takes a task structure pointer and returns the task ID of that task.

5.3.5 Context switching

Angel maintains context blocks for each task under its control through the life of the task, and saves the value of all current processor registers when a task switch occurs. It uses two groups of register context save areas:

- The Angel global register blocks. These are used to store the CPU registers for a task when events such as interrupt and task deschedule events occur.
- An array of available *Task Queue Items* (TQIs). Each allocated TQI contains the information Angel requires to correctly schedule a task, and to store the CPU registers for a task when required.

The global register blocks: angel_GlobalRegBlock

The Angel global register blocks are used by all the exception handlers and the special functions `Angel_Yield()` and `Angel_Wait()`. Register blocks are defined as an array of seven elements. Table 5-2 shows the global register blocks.

Table 5-2 Global register blocks

Register block	Description
RB_Interrupted	Used by the FIQ and IRQ exception handlers.
RB_Desired	Used by `Angel_SerialiseTask()`.
RB_SWI	Saved on entry to a complex SWI and restored on return to the application.
RB_Undef	Saved on entry to the undefined instruction handler.
RB_Abort	Saved on entry to the abort handler.
RB_Yield	Used by the `Angel_Yield()` and `Angel_Wait()` functions.
RB_Fatal	Used only in a debug build of Angel. It saves the context where a fatal error occurred.

In the case of RB_SWI and RB_Interrupted, the register blocks contain the previous task register context so that the interrupt can be handled. If the handler function calls `Angel_SerialiseTask()`, the global register context is saved into the current task TQI.

In the case of RB_Yield, the register block is used to store temporarily the context of the calling task, prior to entering the serializer. The serializer saves the contents of RB_Yield to the TQI entry for the current task, if required.

The Angel task queue: angel_TQ_Pool

The serializer maintains a task queue by linking together the elements of the `angel_TQ_Pool` array. The task queue must contain an idle task entry. Each element of the array is a TQI. A TQI contains task information such as:

- the register context for the task
- the current priority of the task
- the type of the task (for example, TP_Application)
- the task state (for example, TS_Blocked)
- the initial stack value for the task
- a pointer to the next lower-priority task.

The elements in the `angel_TQ_Pool` array are managed by routines within the serializer and must not be modified externally.

Angel calls `Angel_NewTask()` to create new tasks. This function initializes a free TQI with the values required to run the task. When the task is selected for execution, `Angel_SelectNextTask()` loads the register context into the CPU. The context is restored to the same TQI when:

- `Angel_SerialiseTask()` is called as the result of exception processing or a call to `Angel_Yield()`
- `Angel_Wait()` determines that the task must be blocked.

When the debugger requests information about the state of the application registers, the Angel debug agent retrieves the register values from the TQI for the application. The application TQI is updated from the appropriate global register block when exceptions cause Angel code to be run.

Overview of Angel stacks for each mode

The serialization mechanism described in *Angel task management* on page 5-28 ensures that only one task ever executes in Supervisor mode. Therefore, all Angel Supervisor mode tasks share a single stack, on the basis that:

- it is always empty when a task starts
- when the task returns, all information that was on the stack is lost.

The application uses its own stack, and executes in either User or Supervisor mode. Callbacks due to application requests to read or write from devices under control of the Device Driver Architecture execute in User mode, and use the application stack.

The following Angel stacks are simple stacks exclusively used by one thread of control. This is ensured by disabling interrupts in the corresponding processor modes:

- IRQ stack
- FIQ stack
- UND stack
- ABT stack.

The User mode stack is also split into two cases, because the Application stack and Angel stack are kept entirely separate. The Angel User mode stack is split into array elements that are allocated to new tasks, as required. The application stack must be defined by the application.

5.3.6 Example of Angel processing: a simple IRQ

This section gives an example of processing a simple IRQ from start to finish, and describes in more detail how Angel task management affects the receipt of data through interrupts. Refer also to *Angel communications architecture* on page 5-41 for an overview of Angel communications.

Figure 5-4 on page 5-35 shows the application running, when an IRQ is made that completes the reception of a packet.

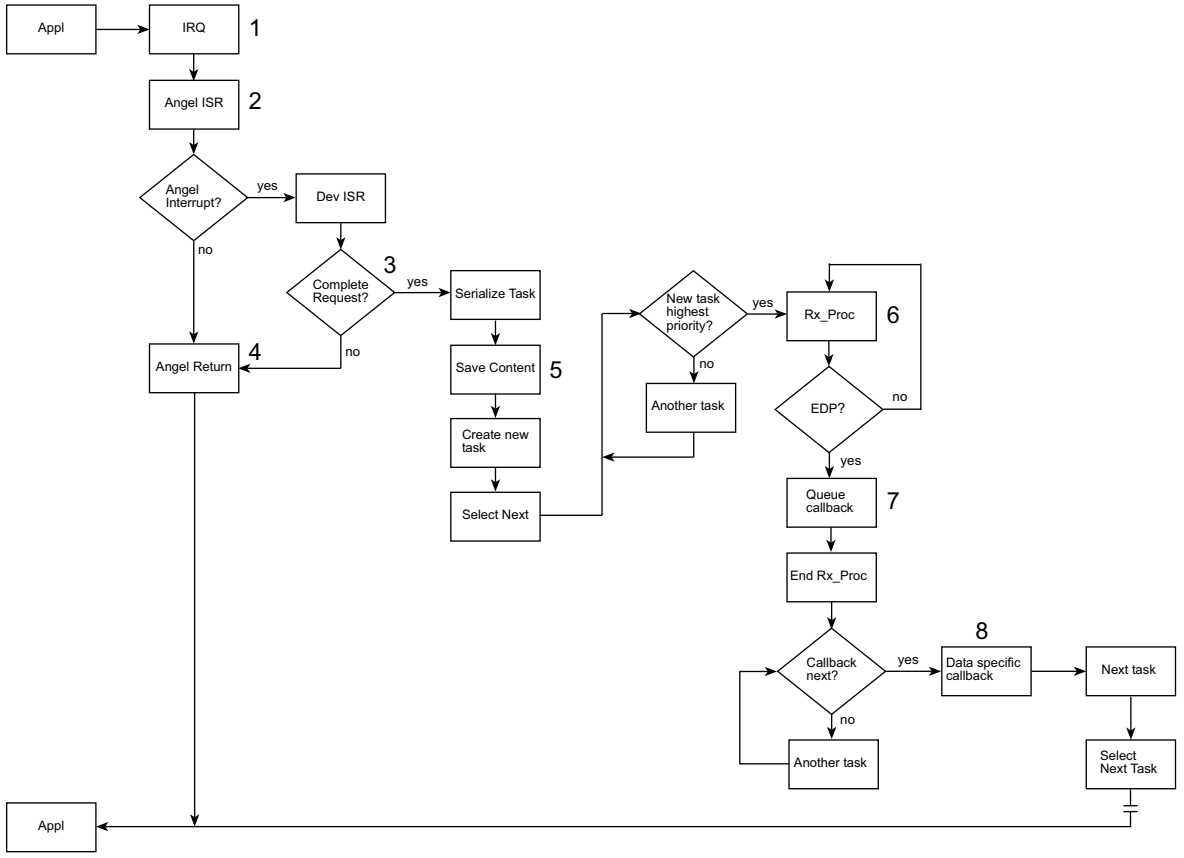

Figure 5-4 Processing a simple IRQ

The IRQ is handled as follows:

1. The Interrupt exception is noticed by the processor. The processor:
   - fetches the IRQ vector
   - enters Interrupt mode
   - starts executing the Angel Interrupt Service Routine.

   On entry to the IRQ handler, FIQ interrupts are disabled if `HANDLE_INTERRUPTS_ON_FIQ=1` (the default is 0, FIQ interrupts enabled). Interrupts are not re-enabled until either:
   - `Angel_SerialiseTask()` is called
   - the interrupt completes.

2. The Angel ISR saves the processor state in a register block, uses the `GETSOURCE` macro to determine the interrupt source, and jumps to the handler. The processor state is saved because this data is required by `Angel_SerialiseTask()`.

3. The interrupt handler determines the cause of the IRQ. If the interrupt is not an Angel interrupt it returns immediately.

   If the interrupt is an Angel interrupt and the driver uses polled input, the handler calls `Angel_SerialiseTask()` to schedule processing. If the driver does not use polled input, the handler calls `Angel_SerialiseTask()` to schedule processing if:
   - the end of packet character is reached
   - the end of request is reached for a raw device (determined by length)
   - the ring buffer is empty (tx), or full (rx).

4. If `Angel_SerialiseTask()` is not required, the ISR reads out any characters from the interrupting device and returns immediately.

5. `Angel_SerialiseTask()` saves the stored context from step 2 and creates a new task. It then executes the current highest priority task. The new task is executed after all tasks of higher priority have been executed.

6. The new task executes in Supervisor mode. It reads the packet from the device driver to create a proper ADP packet from the byte stream.

7. When the packet is complete, the task schedules a callback task to process the newly arrived packet.

8. The callback routine processes the packet and terminates. `Angel_NextTask()` finds that the application is the highest priority task, and `Angel_SelectNextTask()` restarts the application by loading the context stored at step 2 into the processor registers.

5.4 Configuring Angel

This section describes some of the major configuration changes that you can make to Angel. All the configuration changes described in this section are static. You must recompile Angel to implement these changes.

The changes you can make are described in the following sections:

- *Configuring the memory map* on page 5-37
- *Configuring timers and profiling* on page 5-38
- *Configuring exception handlers* on page 5-38
- *Configuring where Angel runs* on page 5-38
- *Configuring SWI numbers* on page 5-40.

5.4.1 Configuring the memory map

You can configure the Angel stack positions by editing the value of:

```
#define Angel_StacksAreRelativeToTopOfMemory
```

in `devconf.h`.

By default, the Angel stacks are configured relative to the top of memory. This is the recommended option. If Angel stacks are configured to start relative to the top of memory then the Angel code searches for the top of contiguous memory and the stack pointers are set at this location. This means that you can add memory to your system without updating the memory map and rebuilding Angel.

You must define the memory map to allow the debugger to control illegal read/writes using the checks in the `PERMITTED` macro. These should reflect the permitted access of the system memory architecture. You must take care with systems that have access to the full 4GB of memory, because the highest section of memory should equate to `0xffffffff` when the base and size are defined as a sum, and it might wrap around to 0.

For example, if there is memory-mapped I/O at `0xffd00000` the definition should be:

```
#define IOBase (0xFFD00000)
#define IOSize (0x002fffff)
#define IOTop  (IOBase + IOSize)
```

not:

```
#define IOBase (0xFFD00000)
#define IOSize (0x00300000)
#define IOTop  (IOBase + IOSize)
```

5.4.2 Configuring timers and profiling

The exact configuration procedure will depend on the development board you are using. The ARM development board, for example, has two timers available, and by default, profiling and Ethernet are configured to use the same timer. The ARM development board uses pc sampling for profiling. This requires a fast interrupt. The interrupt service routine records where the program was when it was interrupted. If you do not use profiling or Ethernet you can use the timer for your application.

You can turn off profiling by setting a runtime debugger variable, but this does not free the timer. In the Angel development board port, profiling is specified in the PROFILE entry of `devconf.h`. You must recompile Angel to remove profiling support.

System timers can be initialized by implementing the `INITTIMER` macro in `target.s`. This macro is not implemented by the ARM development board port. It is provided as a place holder to enable you to initialize your own system timers.

5.4.3 Configuring exception handlers

You can chain your own exception handlers to the Angel exception handlers. Refer to *Chaining exception handlers* on page 5-19 for more information.

5.4.4 Configuring where Angel runs

This section describes how to configure Angel to run from:

- ROM
- ROM mapped to address zero
- RAM (the default).

Link addresses

The makefile for `angel.rom` contains two makefile macros that control the addresses where Angel is linked:

`RWADDR` This defines the base address for read/write areas, such as `dataseg` and `bss` (zero-initialized) areas, along with some assembler areas. Angel requires approximately 24KB of free RAM for its read/write areas.

`ROADDR` This defines the base address for read-only areas. In general, read-only areas are code areas. Angel requires between 50 and 100KB of RAM for its read-only areas.

The target-specific configuration file `devconf.h` contains a number of macros that define the memory layout of the target board. It also contains checks to ensure that the values of `RWADDR` and `ROADDR` are sensible.

Most of these macros are only used within devconf.h (for the sanity checks, in the READ/WRITE_PERMITTED macros, and for defining application stack and heap areas). In addition, the macro ROMBase is used during startup to calculate the offset between the code currently executing in ROM and its eventual ROADDR destination.

ROM locations

Angel supports two types of ROM system:

- ROM mapped to address 0 on reset, and mapped out to RAM during Angel bootstrap
- ROM permanently mapped to address 0.

For the first type:

1. Define ROMBase in devconf.h as the normal (mapped-out) address of the ROM.
2. Set the ROM-only build variable in target.s to FALSE.
3. Provide an assembler macro called UNMAPROM in target.s that maps the ROM away from 0.
4. Declare the makefile macro FIRST as 'startrom.o(ROMStartup)', including the quote (') characters.

For the second type:

1. Define ROMBase in devconf.h as 0.
2. Set the ROMonly build variable in target.s to TRUE.
3. Declare the makefile macro FIRST as 'except.o(__Vectors)', including the single quote (') characters.

Processor exception vectors

Regardless of where you declare RWADDR and ROADDR to be, the ARM processor requires the exception vector table to be located at zero. There are a number of situations where this happens by default, for example when RWADDR is set to 0, or in ROM-at-zero systems.

When this does not happen by default, Angel explicitly copies everything in AREA __Vectors from RWADDR to zero. All code within the __Vectors area must be position-independent, because it is linked to run at ROADDR, not zero.

In most configurations, Angel is able to detect a branch through zero by application code, and report it as an error. However, this is not possible in ROM-at-zero systems. In this case, a branch through zero causes:

- a system reboot if the processor is executing in a privileged mode
- a system crash if the processor is not executing in a privileged mode.

5.4.5 Configuring SWI numbers

Angel requires one SWI in order to operate. The SWI is used to:

- change processor mode to gain system privileges
- make semihosting requests
- report an exception to the debugger.

The SWI passes a reason code in r0 to determine the type of request. Depending on the SWI, additional arguments are passed in r1. Refer to *Input/Output SWIs* on page 6-10 for more information.

The SWI number is different for ARM state and Thumb state. By default, the SWI numbers used are:

ARM state `0x123456`

Thumb state `0xab`

If you want to use either of these SWI numbers for your system you can reconfigure the SWI to use any of the available SWI numbers. If you change these values you must:

- specify the new SWI value in the Angel definition files.
- recompile the debug agent using the new value.

In C, the SWI numbers are defined in `Angel\Source\arm.h` as:

```
#define angel_SWI_ARM (0x123456)
#define angel_SWI_THUMB (0xAB)
```


5.5 Angel communications architecture

This section gives an overview of the Angel communications architecture. It describes how the various parts of the architecture fit together, and how debugging messages are transmitted and processed by Angel. For full details of the Angel Debug Protocol and messages, refer to *Angel Debug Protocol* and *Angel Debug Protocol Messages*.

5.5.1 Overview of the Angel communications layers

Figure 5-5 shows a conceptual model of the communication layers for Angel. In practice, some layers might be combined.

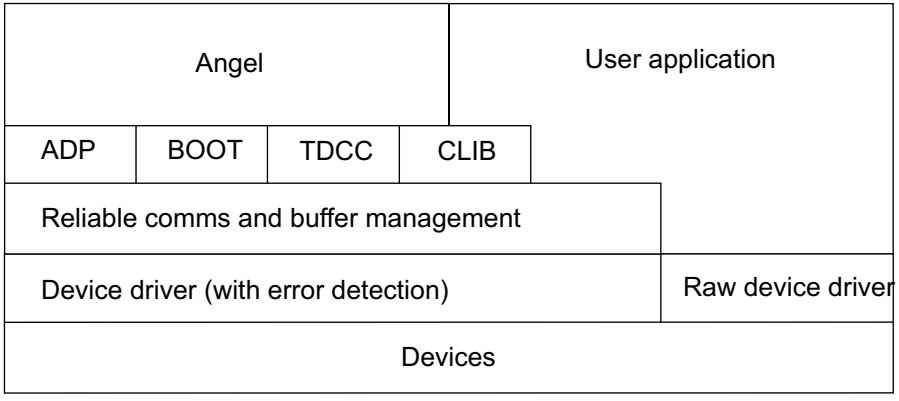

Figure 5-5 Communications layers for Angel

The channels layer includes:

ADP The Angel Debug Protocol channel. This consists of the *Host ADP channel* (HADP) and *Target ADP channel* (TADP).

BOOT The boot channel.

TDCC The Thumb debug communications channel.

CLIB C library support.

At the top level on the target, the Angel agent communicates with the debugger host, and the user application can make use of semihosting support (CLIB).

All communications for debugging (ADP, BOOT, TDCC, CLIB) require a reliable channel between the target and the host. The reliable communications and buffer management layer is responsible for providing reliability, retransmissions, and multiplexing/demultiplexing for these channels. This layer must also handle buffer management, because reliability requires retransmission after errors have occurred.

The device driver layer detects and rejects bad packets but does not offer reliability itself.

5.5.2 Boot support

If there are two or more debug devices (for example, serial and serial/parallel), the boot agent must be able to receive messages on any device and then ensure that further messages that come through the channels layer are sent to the correct (new) device.

When the debug agent detects a Reboot or Reset message, it listens to the other channels using the device that received the message. All debug channels switch to use the newly selected debug device.

During debugging, each channel is connected through the same device to one host. Initially, Angel listens on all Angel-aware devices for an incoming boot packet, and when one is received, the corresponding device is selected for further Angel use. Angel listens for a reset message throughout a debugging session, so that it can respond to host-end problems or restarts.

To support this, the channels layer provides a function to register a read callback across all Angel-aware devices, and a function to set the default device for all other channel operations.

5.5.3 Channels layer and buffer management

The channels layer is responsible for multiplexing the various Angel channels onto a single device, and for providing reliable communications over those channels. The channels layer is also responsible for managing the pool of buffers used for all transmission and reception over channels. Raw device I/O does not use the buffers.

Although there are several channels that could be in use independently (for example, CLIB and HADP), the channel layer accepts only one transmission attempt at a time.

Channel restrictions

To simplify the design of the channels layer and to help ensure that the protocols operating over each channel are free of deadlocks, the following restriction is placed on the use of each channel.

For a particular channel, all messages must originate from either the Host or the Target, and responses can be sent only in the opposite direction on that channel. Therefore two channels are required to support ADP:

- one for host-originated requests (Read Memory, Execute, Interrupt Request)
- one for target-originated requests (Thread has stopped).

Each message transmitted on a channel must be acknowledged by a reply on the same channel.

Buffer management

Managing retransmission means that the channels layer must keep messages that have been sent until they are acknowledged. The channel layer supplies buffers to channel users who want to transmit, and then keeps transmitted buffers until acknowledged.

The number of available buffers might be limited by memory to less than the theoretical maximum requirement of one for each channel and one for each Angel-aware device.

The buffers contain a header area sufficient to contain channel number and sequence IDs, for use by the channels layer itself. Any spare bits in the channel number byte are reserved as flags for future use.

Long buffers

Most messages and responses are short (typically less than 40 bytes), although some can be up to 256 bytes long. However, there are some situations where larger buffers would be useful. For example, if the host is downloading programs or configuration data to the target, a larger buffer size reduces the overhead created by channel and device headers, by acknowledgment packets and by the line turnaround time required to send each acknowledgment (for serial links). For this reason, a long (target-defined but suggested size of 4KB) buffer is available for target memory writes, that are used for program downloads.

Limited RAM

When RAM is unlimited, the easiest solution is to make all buffers large. There is a mechanism that allows a single large buffer to be shared, because RAM in an Angel system is not normally an unlimited resource.

When the device driver has read enough of a packet to determine the size of the packet being received, it performs a callback asking for a suitably sized buffer. If a small buffer is adequate, a small buffer is provided. If a large buffer is required, but is not available, the packet is treated as a bad packet, and a resend request results.

Buffer life cycle

When sending data, the user of a channel must explicitly allocate a buffer before requesting a write. Buffers must be released *either* by:

- Passing the buffer to one of the channel transmit functions in the case of reliable data transmission. In this case, the channels code releases the buffer.
- Explicitly releasing it with the release function in the case of unreliable data transmission.

Receive buffers must be explicitly released with the release function (Figure 5-6).

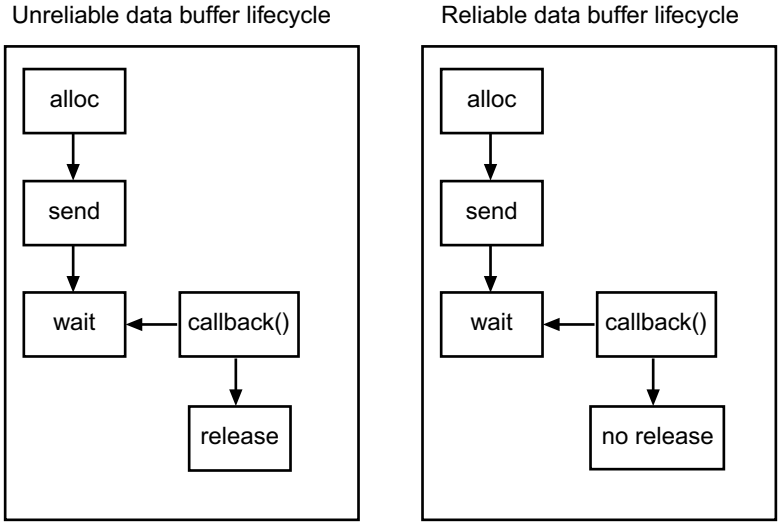

Figure 5-6 Send buffer lifecycle

Channel packet format

Channel packets contain information, including:

- channel ID, such as the HADP ID
- packet number
- acknowledged packet number
- flags used for distinguishing data from control information.

Refer to *Angel Debug Protocol* for a complete description of the channel packet format.

The length of the complete data packet is returned by the device driver layer. An overall length field for the user data portion of the packet it not required, because the channel header is fixed length.

Heartbeat mechanism

Heartbeats must be enabled for reliable packet transmission to work.

The remote_a heartbeat software writes packets using at least the heartbeat rate, and uses heartbeat packets to ensure this. It expects to see packets back using at least the packet timeout rate, and signals a timeout error if this is violated.

5.5.4 Device driver layer

Angel supports polled and asynchronous interrupt-driven devices, and devices that start in an asynchronous mode and finish by polling the rest of a packet. At the boundary of the device driver layer, the interface offers asynchronous (by callback) read and write interfaces to Angel, and a synchronous interface to the application.

Support for callback across all devices

This is primarily a channels layer issue, but because the boot channel must listen on all Angel-compatible devices, the channels layer must determine how many devices to listen to for boot messages, and which devices those are.

To provide this statically, the devices layer exports the appropriate device table or tables, together with the size of the tables.

Transmit queueing

Because the core operating mode is asynchronous and more than one thread can use a device, Angel rejects all but the first request, returns a `busy` error message, and leaves the user (channels or the user application) to retry later.

Angel interrupt handlers

Angel interrupt handlers are installed statically, at link time. The Angel interrupt handler runs off either IRQ or FIQ. It is recommended that it is run off IRQ. The Angel interrupt is defined in `devconf.h`.

Control calls

Angel device drivers provide a control entry point that supports the enable/disable transmit/receive commands, so that Angel can control application devices at critical times.

5.6 The Fusion IP stack for Angel

The Ethernet hardware is specific to the development board you are using. The sections below assume a ARM development board, Fusion IP stack, and Ethernet Adaptor Kit (No. KPI 0014D).

5.6.1 How Angel, Fusion, and the ARM development board hardware fit together

The Ethernet interface for the ARM development board card is provided by an Olicom EtherCom PCMCIA Ethernet card installed in either PCMCIA slot. The Olicom card uses an Intel i82595 Ethernet controller.

The UDP/IP stack is the Pacific Softworks Fusion product, ported to ARM and the Angel environment. The drivers for PCMCIA and the Ethernet card have been implemented, as has the Angel device driver to make the whole stack appear as an Angel device. Figure 5-7 shows how the components fit together.

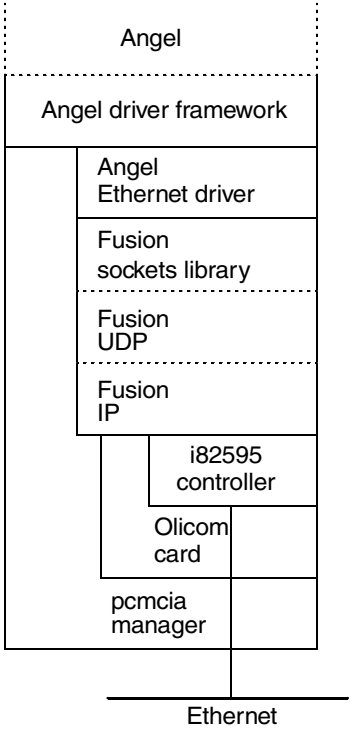

Figure 5-7 Angel, Fusion, and ARM development board hardware

Initialization

The stack is initialized in the following sequence:

1. `devclnt.c:angel_InitialiseDevices()` calls `ethernet.c:ethernet_init()` to open a socket.
2. `fusion:socket()` notices that the fusion stack has not been initialized. Fusion stack initialization calls:
   a. `olicom.c:olicom_init()` calls:
   b. `pcmcia.c:pcmcia_setup()` detects Olicom card and calls:
   c. `olicom.c:olicom_card_handler()` with a card insertion event and then:
   d. `olicom.c:read_card_params()` to register `olicom_isr()` with `pcmcia.c`.
3. Fusion stack initialization calls:

   `olicom.c:olicom_updown()` and, through `olicom_state()`:

   `82595.c:i595_bringup()` to complete the initialization sequence.

Angel Ethernet device driver

After initialization, the Angel side of the driver is implemented as a polling device. At every call to `Angel_Yield()`, `angel_EthernetPoll()` is invoked, and non-blocking `recv()` calls are made to the Fusion stack to see if data is waiting on any of the sockets.

Outgoing packets are passed to the Fusion stack in a single step by calling `sendto()`.

Interrupt handling

The bottom of the Fusion stack is driven by interrupts from the Olicom card. Interrupts are handled in the following sequence:

1. `suppasm.s:angel_DeviceInterruptHandler()` calls the `GETSOURCE` macro in `pid/target.s` to identify the PCMCIA controller as the source.
2. `pcmcia.c:angel_PCMCIAIntHandler()` establishes that it is an I/O interrupt and calls the routine registered during initialization.
3. `olicom.c:olicom_isr()` checks the interrupt, switches off interrupts from the Olicom card, and serializes `olicom_process()` to do the processing with all other interrupts enabled.
4. `olicom.c:olicom_process()` identifies the reason for the interrupt and passes it as an event to `olicom_state()`.
5. `olicom.c:olicom_state()` calls an appropriate routine in `82595.c` to handle packet reception and transmission.
6. `82595.c` routines control the i82595 chip and transfer packets in both directions between Fusion buffers and the chip. Calls are made to Fusion functions as appropriate.
7. `olicom.c:olicom_process()` checks to see whether all interrupt events have been serviced. If so, Olicom interrupts are re-enabled. If not, `olicom_process()` queues itself again and then exits in case another device is waiting for the serializer lock.

Additionally, the Fusion stack can make calls to `olicom_start()` (to queue a new packet for transmission), `olicom_ioctl()`, and `olicom_updown()` in response to socket calls from the Angel Ethernet driver or as a result of packet processing.

Chapter 6
Semihosting SWIs

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:

6.1 Overview of the C library support SWIs

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.

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 Tools Guide* for more information on support for semihosting in the ARM C library.

Figure 6-1 shows an overview of semihosting.

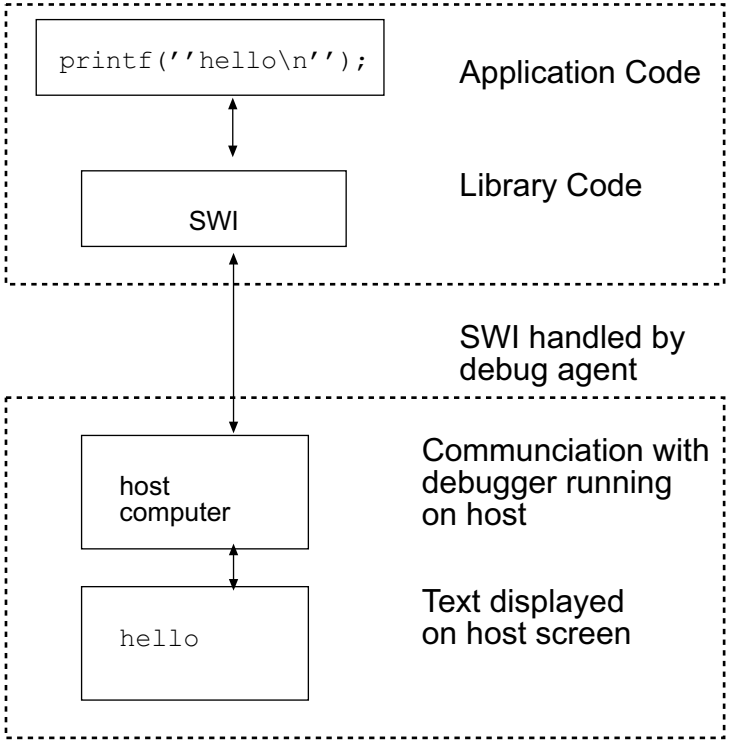

Figure 6-1 Semihosting overview

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

6.1.1 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 handlers.

Semihosting operations are requested using a single SWI number in order to leave the other SWI numbers available for use by the application or operating system. By default the SWI used for semihosting is:

`0x123456` in ARM state

`0xAB` in Thumb state

You can configure the semihosting SWI number, but this is not advisable because you must then ensure that all code in your system, including library code, uses the new SWI number.

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 system call is made.

The available semihosting operation numbers for operation type are allocated as follows:

`0x00` to `0x31`
: This is used by ARM.

`0x32` to `0xff`
: This is reserved for future use by ARM.

`0x100` to `0x1ff`
: This is reserved for user applications. They will not be used by ARM.

  If you are writing your own SWI operations, however, you should use a different SWI number rather than using the semihosted SWI number and these operation type numbers.

`0x200` to `0xffffffff`
: This is undefined. They are not currently used and not recommended for use.

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 `arm.h`. You can define the operation names with an `EQU` directive. For example:

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

6.2 Semihosting implementation

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

6.2.1 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, you should set the variable `OS` in your `armul.cnf` file to `None`.

See the *Peripheral models* on page 2-26 for more details.

6.2.2 Angel

Angel installs a SWI handler 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.

6.2.3 Multi-ICE and EmbeddedICE

When using a protocol convertor such as Multi-ICE, in default configuration, or the EmbeddedICE interface, semihosting is implemented as follows:

1. A breakpoint is set on the SWI vector.
2. When this breakpoint is hit, the protocol convertor examines the SWI number.
3. If the SWI is recognized, the protocol convertor emulates it and transparently restarts execution of the application.

   If the SWI is not recognized as a semihosting SWI, the protocol convertor halts the processor and reports an error.

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

`$semihosting_enabled`

By default, this variable is set to 1 to enable semihosting. Setting it to 0 disables semihosting. This can be useful, for example, when debugging applications running from ROM. Disabling semihosting in such situations frees up another watchpoint unit. Enable semihosting by setting `$semihosting_enabled` rather than setting the S bit in `$vector_catch`.

`$semihosting_vector`

This variable controls the location of the breakpoint set by the protocol converter to detect a semihosted SWI. It is set to to the SWI entry in the exception vector table (`0x8`) by default. The protocol converter handles the semihosted SWI and then examines the contents of lr and returns to the instruction following the SWI instruction in your code. This completely bypasses the contents of the `$semihosting_vector` address.

If this variable is set to 0, this does not imply address 0. Address 8 is used instead. 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.

6.2.4 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.

6.3 Adding an application SWI handler

In some circumstances it is useful to have both the semihosted SWIs and your own application-specific SWIs available. In such cases you must ensure that the two SWIs cooperate correctly. The way to ensure this depends upon the debug agent in use.

6.3.1 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, the semihosting functionality in ARMulator detects that it is not a semihosting SWI and executes the instruction in the SWI entry of the vector table instead. This instruction branches to your own SWI handler.

6.3.2 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*.

6.3.3 Multi-ICE and EmbeddedICE

To ensure that the application SWI handler will successfully cooperate with the protocol convertor 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.

At the point Multi-ICE or EmbeddedICE interface traps the SWI, your own SWI handler must have already restored all registers to the values when your SWI handler was entered. Typically, this means that your SWI handler should 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.

For example, a particular SWI handler can detect if it has failed to handle a SWI and branch 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, r0              ; ...and restore it.
    LDMFD sp!, {r0-r12,pc}^   ; Restore registers and return.
```

This code could be modified for use with Multi-ICE or EmbeddedICE interface semihosting as follows:

```
; r0 = 1 if SWI handled
    CMP r0, #1                ; Test if SWI has been handled.
    LDMFD sp!, {r0}           ; Unstack SPSR...
    MSR spsr, 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 /
                              ; EmbeddedICE interface handler.
```

The `$semihosting_vector` variable should be set up to point to the address of `Semi_SWI`. The instruction at `Semi_SWI` never gets executed because the protocol converter returns directly to the application after processing the semihosted SWI (see Figure 6-2).

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.

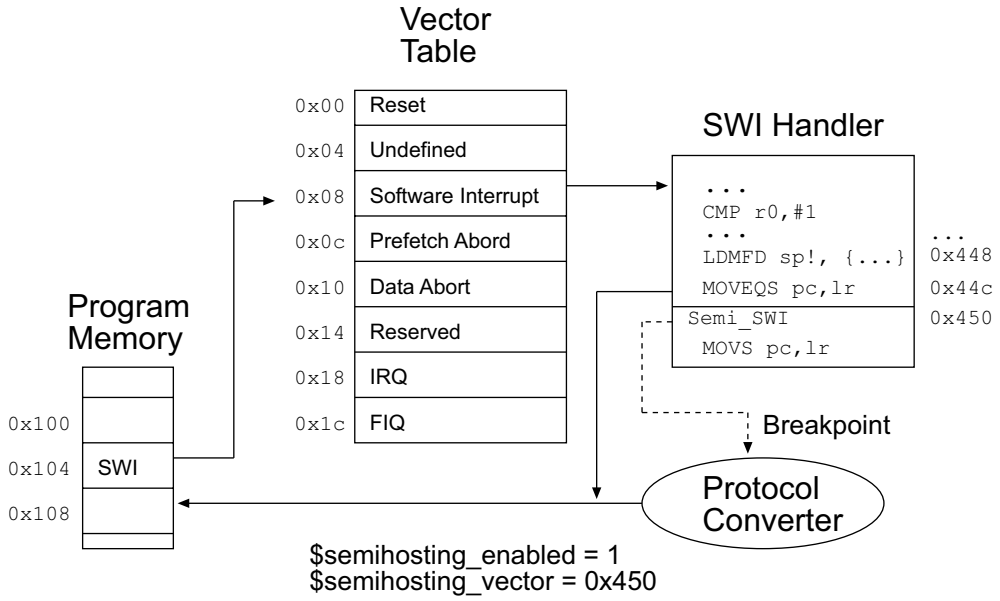

Figure 6-2 Semihosting with breakpoint

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 `$semihosting_vector` address has moved to a place that cannot currently be reached, the protocol convertor no longer knows about the triggered breakpoint. To prevent this from happening, you should change the contents of `$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.

6.3.4 Multi-ICE DCC semihosting

When using the DCC semihosting mechanism, adding an application SWI handler should be done in exactly the same way as non-DCC semihosting (see *Multi-ICE and EmbeddedICE* on page 6-7).

6.4 Input/Output SWIs

The SWIs listed in Table 6-1 implement the semihosted input/output operations. These operations are used by C library functions such as `printf()` and `scanf()`. They can be treated as an ATPCS function call. However, except for r0 that contains the return status, they restore the registers they are called with before returning.

Table 6-1 Semihosting SWIs

SWI	Description
SYS_OPEN (0x01) on page 6-11	Open a file on the host.
SYS_CLOSE (0x02) on page 6-12	Close a file on the host.
SYS_WRITEC (0x03) on page 6-12	Write a character to the console.
SYS_WRITE0 (0x04) on page 6-13	Write a null-terminated string to the console.
SYS_WRITE (0x05) on page 6-13	Write to a file on the host.
SYS_READ (0x06) on page 6-14	Read the contents of a file into a buffer.
SYS_READC (0x07) on page 6-14	Read a byte from the console.
SYS_ISERROR (0x08) on page 6-15	Determine if a return code is an error.
SYS_ISTTY (0x09) on page 6-15	Check whether a file is connected to an interactive device.
SYS_SEEK (0x0a) on page 6-16	Seek to a position in a file.
SYS_FLEN (0x0c) on page 6-16	Return the length of a file.
SYS_TMPNAM (0x0d) on page 6-17	Return a temporary name for a file.
SYS_REMOVE (0x0e) on page 6-17	Remove a file from the host.
SYS_RENAME (0x0f) on page 6-18	Rename a file on the host.
SYS_CLOCK (0x10) on page 6-18	Number of centiseconds since execution started.
SYS_TIME (0x11) on page 6-19	Number of seconds since Jan 1, 1970.
SYS_SYSTEM (0x12) on page 6-19	Pass a command to the host command-line interpreter.
SYS_ERRNO (0x13) on page 6-19	Get the value of the C library `errno` variable.
SYS_GET_CMDLINE (0x15) on page 6-20	Get the command-line used to call the executable.

Table 6-1 Semihosting SWIs (Continued)

SWI	Description
SYS_HEAPINFO (0x16) on page 6-21	Get the system heap parameters.
SYS_ELAPSED (0x30) on page 6-22	Get the number of target ticks since execution started.
SYS_TICKFREQ (0x31) on page 6-22	Define a 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.

6.4.1 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 debuggers 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 6-2 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 6-2 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.

6.4.2 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.

6.4.3 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.

6.4.4 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.

6.4.5 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.

The file operation should be performed as a single action whenever possible. That is, a write of 16KB should not be split 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.

6.4.6 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. The file operation should be performed as a single action whenever possible. That is, a read of 16KB should not be split 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 6-2 on page 6-11 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.

6.4.7 SYS_READC (0x07)

Reads a byte from the debug channel. The read is notionally from the keyboard attached to the debugger.

Entry

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

Return

On exit, r0 contains the byte read from the debug channel.

6.4.8 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.

6.4.9 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.

6.4.10 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.

6.4.11 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.

6.4.12 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

word 3 This contains the length of the buffer. The length should 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.

6.4.13 SYS_REMOVE (0x0e)

Deletes a specified file.

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.

6.4.14 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.

6.4.15 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 the Multi-ICE debug agent the request is passed back to the host for execution. This can lead to unpredictable delays in transmission and process scheduling.

This function should be used only to calculate time intervals (the length of time some action took) by calculating the difference in the result on two occasions.

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).

6.4.16 SYS_TIME (0x11)

Returns the number of seconds since 00:00 January 1, 1970.

Entry

There are no parameters. Register r1 must contain zero.

Return

On exit, r0 contains the number of seconds.

6.4.17 SYS_SYSTEM (0x12)

Passes a command to the host command-line interpreter. This enables you to execute a system command such as `ls`, or `pwd`. The terminal I/O is on the host, and is not visible to the target.

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.

6.4.18 SYS_ERRNO (0x13)

Returns the value of the C library `errno` variable associated with the host support for the debug monitor. 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 or not, and to what value `errno` is set is completely 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.

6.4.19 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).

6.4.20 SYS_HEAPINFO (0x16)

Returns the system heap parameters. The values returned are typically those used by the C library during initialization. These values are defined in the `devconf.h` header file. For Multi-ICE, the values returned are the image location and the top of memory.

The C library can override these values, but will do so only if `__heap_base` is defined at link time (see *ADS Tools Guide* for more information on memory management in the C library). In this case the values of the following symbols are used:

- `__heap_base`
- `__heap_limit`
- `__stack_base`
- `__stack_limit`

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 6-1 for the structure of the data block and return values.

Example 6-1

```
struct block2 {
    int heap_base;
    int heap_limit;
    int stack_base;
    int stack_limit;
    }
struct block2 *mem_block, info;
mem_block = & info;
SemiSWI(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 6-2.

Example 6-2

```
Heap Base = 0x00000000
Heap Limit = 0x00076e00
Stack Base = 0x00078e00
Stack Limit = 0x00076e00
```

6.4.21 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. This follows the convention used by the ARM compilers for the **`long long`** data type.

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.

6.4.22 SYS_TICKFREQ (0x31)

Defines a tick frequency.

Entry

On entry, r0 contains the reason code `0x31`

Exit

On exit, r0 contains either:

- the ticks per second
- –1 if the target does not know the value of one tick.

6.5 Debug agent interaction SWIs

In addition to the C library semihosted functions described in *Input/Output SWIs* on page 6-10, 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.

6.5.1 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. Under Angel, 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 or Multi-ICE:

- 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

On entry, r0 contains `0x17`. 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 should 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 should be pushed onto the stack before the call and popped afterwards, as for a `BL` function call.

6.5.2 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 r0 is set to `Angel_SWIreason_ReportException` and r1 is set to one of the values listed in Table 6-3 and Table 6-4 on page 6-25. These values are defined in `adp.h`.

`ADP_UserInterruption` is generated by Angel if the debugger sends an `ADP_InterruptRequest` to stop the application. `ADP_Breakpoint` is generated when Angel detects attempted execution of a breakpoint instruction. Angel does not implement watchpoints, although other debug agents do.

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 6-3 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

Table 6-4 Software reason codes

Name (#defined in adp.h)	Hexadecimal value
ADP_Stopped_BreakPoint	0x20020
ADP_Stopped_WatchPoint	0x20021
ADP_Stopped_StepComplete	0x20022
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 6-4 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.

6.5.3 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.

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.

ARMulator ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.

armsd The *ARM Symbolic Debugger* (armsd) is an interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language. It is a command-line debugger that runs on all supported platforms.

AXD See *ARM eXtended Debugger*.

Basic ARM Ten System The *Basic ARM Ten System* (BATS) is a modelling scheme similar to but separate from *ARMulator*. BATS is designed specifically to model systems based on the ARM10 processor. *ARMulator* models systems based on all earlier ARM processors.

BATS See *Basic ARM Ten System*.

Big-endian Memory organization where the least significant byte of a word is at a higher address than the most significant byte. See also *Little-endian*.

Breakpoint A location in the image. If execution reaches this location, the debugger halts execution of the image. See also *Watchpoint*.

Configuration TRace file (CTR) BATS configuration file. Describes configuration of BATS components, and their interconnections.

Context The information stored in a block of registers on entry to a subroutine, and held there until needed for restoring the information on exit from the subroutine.

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*.

CTR See *Configuration TRace file*

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 needed by an executing program. A small program that helps a larger program communicate with a device such as a printer or keyboard is often packaged as a DLL.

ELF Executable Linkable Format.

Executable image See *Image*.

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 An file of executable code which can be loaded into memory on a target and executed by a processor there.

JTAG Joint Test Access Group. Many debug and programming tools use a JTAG interface port to communicate with processors. For further information refer to IEEE Standard, Test Access Port and Boundary-Scan Architecture specification 1149.1 (JTAG).

Little-endian Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also *Big-endian*.

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.

PID A platform-independent development board designed and supplied by ARM Ltd.

PIE A platform-independent evaluator card designed and supplied by ARM Ltd.

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

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

Saved Program Status Register See *Program Status Register*.

Semihosting A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.

Source File A file which is processed as part of the image building process. Source files are associated with images.

SPSR Saved Program Status Register. See *Program Status Register*.

SWI Software Interrupt. An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.

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.

Thread A thread of execution on a processor.

A context of execution on a processor. A thread is always related to a processor and may or may not be associated with an image.

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*.

Upcall Also called *Callback*. ARMulator models can use upcalls if they need to be informed when state values change.

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.

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.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Z

Symbols

