RealView® ARMulator® ISS

Version 1.4.3

User Guide

ARM®

ARM DUI 0207D

RealView ARMulator ISS
User Guide

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
August 2002	A	Non-Confidential	Release 1.3
	B	Non-Confidential	Not Released
January 2004	C	Non-Confidential	Release 1.4 for RVDS v2.1
March 2007	D	Non-Confidential	Release 1.4.3 for RVDS v3.1

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
RealView ARMulator ISS User Guide

Preface

This preface introduces the *RealView® ARMulator® Instruction Set Simulator* (RVISS) target. It contains the following sections:

- *About this book* on page viii
- *Feedback* on page xi.

Note

Models of the Cortex™ family of processors are provided by the *Instruction Set System Model* (ISSM) software simulator. Details of these models are described in the *RealView Debugger Target Configuration Guide*.

About this book

This book provides reference information for RVISS, the ARM® processor simulator.

Intended audience

This book is written for all ARM developers. It assumes that you are an experienced software developer, and that you are familiar with the ARM development tools provided with RealView Development Suite.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the material in this book, and a summary description of RVISS.

Chapter 2 ***RVISS Basics***

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

Chapter 3 ***Writing RVISS Models***

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

Chapter 4 ***RVISS Reference***

This chapter provides more details to help you use RVISS.

Appendix A ***Using MPCore Models***

This appendix gives details about the MPCore™ model.

Appendix B ***ARM1136JF-S and ARM1136J-S Models***

This appendix gives details about the ARM1136JF-S™ and ARM1136J-S™ models.

Typographical conventions

The following typographical conventions are used in this book:

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

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

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

`mono`space Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`monospace italic`

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

`monospace bold`

Denotes language keywords when used outside example code.

Further reading

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

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

ARM publications

This book contains information that is specific to RVISS. See the RealView Debugger documentation for information on using RVISS with RealView Debugger.

For details on ARM architectures, see:

- *ARMv6-M Architecture Reference Manual* (ARM DDI 0419)
- *ARMv7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)

Feedback

ARM Limited welcomes feedback on both RealView ARMulator ISS, and its documentation.

Feedback on RealView ARMulator ISS

If you have any problems with RealView ARMulator ISS, contact your supplier. To help your supplier provide a rapid and useful response, give:

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

Feedback on this book

If you have any problems with this book, 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 *RealView® ARMulator® Instruction Set Simulator* (RVISS). It contains the following section:

- *RealView ARMulator ISS overview* on page 1-2.

1.1 RealView ARMulator ISS overview

You can debug your prototype software using RealView Debugger which runs on your host computer, and is connected to a target system that runs your prototype software.

Your target system can be any one of:

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

This document describes only the RVISS. For details of the other target systems, see the documentation for that target.

1.1.1 What is RealView ARMulator ISS?

RVISS is an *Instruction Set Simulator* (ISS). It simulates the instruction sets and architecture of ARM processors, together with a memory system and peripherals. You can extend it to simulate other peripherals and custom memory systems.

You can use RVISS for software development and for benchmarking ARM architecture-targeted software. It models the instruction set and counts cycles. There are limits to the accuracy of benchmarking.

See also

- *Accuracy* on page 2-2
- Chapter 3 *Writing RVISS Models*.

1.1.2 Semihosting

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

The ARM C and C++ code use semihosting facilities by default.

To access semihosting facilities from assembly code, use the semihosting *Supervisor Call* (SVC). RVISS intercepts any semihosting SVC call, and then requests service from RealView Debugger.

See also

- *RealView Compilation Tools Libraries and Floating Point Support Guide*.

Chapter 2
RVISS Basics

This chapter describes *RealView® ARMulator® Instruction Set Simulator* (RVISS), a collection of programs that provide software simulation of ARM® processors. It contains the following sections:

Note

The RVISS Profiler feature is not supported by RealView Debugger. However, you can use RealView Debugger tracing to capture profiling information.

See the *RealView Debugger Trace User Guide* for more details.

2.1 About RVISS

RVISS is an instruction set simulator. It simulates the instruction sets and architecture of various ARM processors. To run software on RVISS, you can access it using RealView Debugger.

RVISS is suited to software development and benchmarking ARM architecture-targeted software. It models the instruction set and counts cycles (see *RVISS cycle types* on page 2-16). There are limits to the accuracy of benchmarking and cycle counting, see *Accuracy*.

RVISS provides all the facilities required to enable complete C or C++ programs to run on the simulated system. For information on the C library semihosting SVCs supported by RVISS, see the *RealView Compilation Tools Libraries and Floating Point Support Guide*.

2.1.1 Accuracy

RVISS is not 100% cycle accurate, because it is not based on the actual processor design. In general, models of the less complex, uncached ARM processor cores are reasonably cycle accurate, but models of the cached variants might not correspond exactly with the actual hardware. This assumes that the main memory modelled has a von Neumann architecture and that the cycle accuracy of the main memory is correct.

RVISS is suitable for use as a software development tool for system design, but a hardware model must be used if 100% accuracy is required.

RVISS does not model Asynchronous Mode on cached cores. If you set the control bits in CP15 to specify Asynchronous Mode, RVISS gives a warning:

```
Set to Asynch mode, WARNING this is not supported
```

You can continue debugging, but RVISS behaves exactly as it does in Synchronous Mode.

RVISS memory models do not support multilayer *Advanced High-performance Bus* (AHB). Cores with multiple external buses, such as the ARM926EJ-S™ and ARM11 variants, present a unified bus to the memory system.

For ARM10 and ARM 11 models and Intel XScale technology-based microarchitecture processors, be aware of the following:

- In RealView Debugger tracing, when you view the interleaved instruction and data access trace, then all data accesses appear to be captured within a cycle of the instruction being executed. In reality the data accesses are spread out in time.

2.2 Connections to RVISS in RealView Debugger

RealView Debugger enables you to connect to RVISS models using a RealView Connection Broker interface. RVISS models communicate with RealView Connection Broker through an intermediate interface called *SimRdi_Manager*. See the following documentation for more details on how to connect to RVISS models:

- *RealView Debugger User Guide*
- *RealView Debugger Target Configuration Guide*.

Figure 2-1 is a simplified diagram showing the interaction between RealView Debugger and RVISS through the RealView Connection Broker.

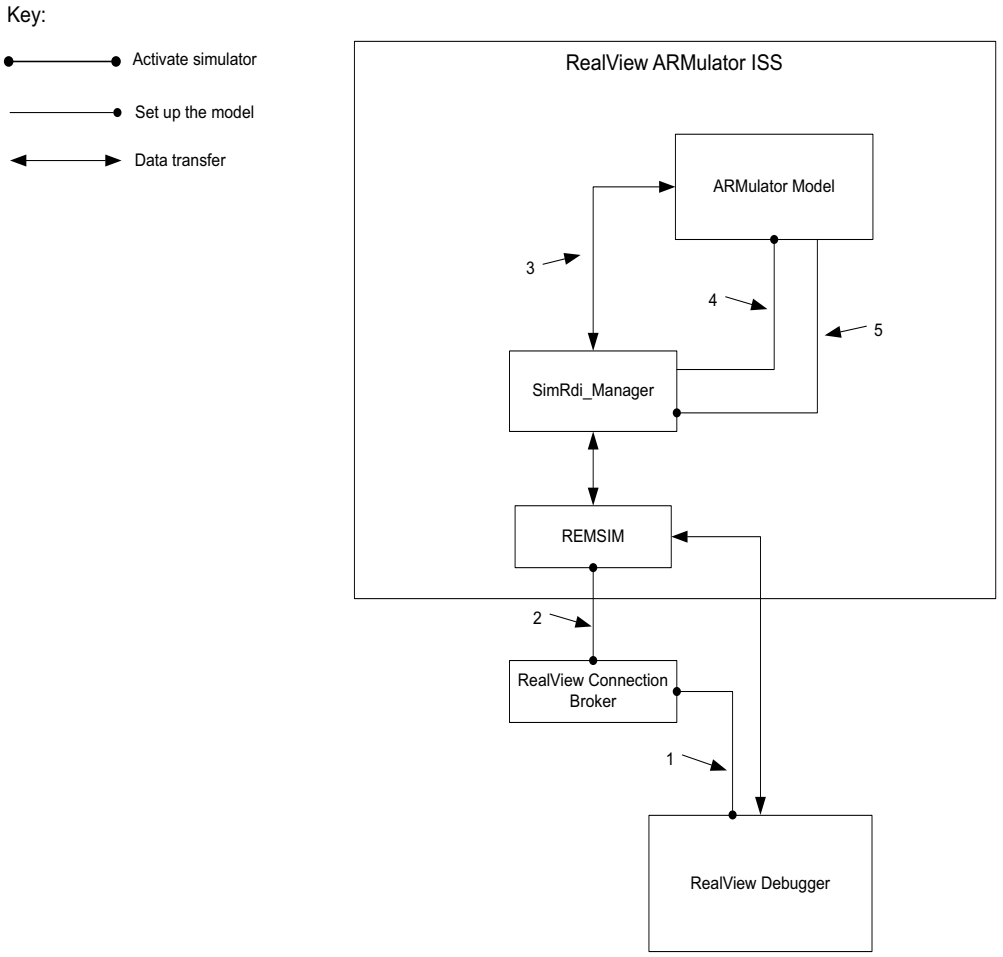

Figure 2-1 RealView Debugger connections to RVISS

In Figure 2-1 on page 2-3, RealView Debugger must first establish a connection to RVISS at the request of the user. The connection is established as follows, and the numbers in parentheses refer to the callouts on the diagram:

1. RealView Debugger contacts RealView Connection Broker (1).
2. RealView Connection Broker activates RVISS through the REMSIM interface (2), and this in turn calls SimRdi_Manager.
3. SimRdi_Manager instantiates the required RVISS model (3).
4. The RVISS model registers with SimRdi_Manager, advertising the SimRdi_Manager services it is providing, and installs a listener to wait for announcements from SimRdi_Manager (4).
5. When the RVISS model setup is finished, requests can be received from, and responses sent to, the RealView Debugger (5).

SimRdi_Manager interface on page 4-3 describes how to write RVISS models that communicate with the SimRdi_Manager interface.

2.2.1 RealView Debugger features supported for RVISS connections

RealView Debugger connections to RVISS models support the following features:

- data trace
- watch points
- ability to connect to multiple instances of RVISS.

2.3 RVISS components

RVISS consists of a series of modules, implemented as *Dynamic Link Libraries* (`.dll` files) for Windows, or as *Shared Objects* (`.so` files for Red Hat Linux, or `.sdi` on all platforms).

The main modules are:

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

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

One of the predefined memory models, `mapfile`, enables you to specify a simulated memory system in detail. It enables you to specify custom memory attributes, such as access width and wait states, for a defined address range.

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

- model additional hardware, such as a coprocessor, peripherals, or memories
- model pre-installed software, such as a C library, semihosting SVC handler, or an operating system
- extract debugging or benchmarking information.

Note

ARM10, ARM11 and Intel XScale technology-based models are not suitable for benchmarking.

You can use different combinations of predefined modules and different memory maps.

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

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

The source code of some modules is supplied. You can use these as examples to help you write your own modules.

2.3.1 Configuring RVISS

You can configure some features of an RVISS model with the RealView Debugger ARMulator Configuration dialog box. However, RVISS provides configuration files with the extension `.ami` that enable you to configure the predefined RVISS modules. The remaining sections in this chapter describe each of the predefined modules, and how you can configure them.

Supplied RVISS configuration files

The following RVISS configuration files are supplied:

- `bustypes.ami`
- `default.ami`
- `example1.ami`
- `peripherals.ami`
- `processors.ami`
- `v6processors.ami`
- `vfp.ami`.

These files are located in the following directory, which is specified in the `ARMCONF` environment variable:

`install_directory``\RVARMulator\ARMulator\...\...\`*`platform`*

In this path:

- *`platform`* is:
  - `win_32-pentium` for Windows
  - `linux-pentium` for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

Modifying the RVISS configuration files

You do not have to edit the supplied RVISS configuration files. You can:

1. Create a new `.ami` file, or edit copies of an existing `.ami` file.
2. Make sure that the path to your `.ami` file is put on your `ARMCONF` environment variable before the path to the standard files. This ensures that your configurations overrides the model parameters.

However, make sure that your file has the same structure as the `.ami` file on which it is based.

If you write any RVISS models of your own, you can produce additional `.ami` files to allow your models to be configured.

Note

Where there is a conflict between configuration settings in a `.ami` file, and settings you have made from in RealView Debugger, the RealView Debugger settings take precedence.

How the configured features are used

When you connect to an RVISS model from RealView Debugger, RVISS reads all the `.ami` files on any of the paths it finds in the `ARMCONF` environment variable. Some features, such as a map file, are available when you connect to a model. For features such as the Tracer, you must enable them in RealView Debugger.

Note

RVISS reads all `.ami` files on these paths. Change the file extension of any back-ups you make of files that you have edited, or copy them to a backup directory. This prevents RVISS reading both the old and new versions. If you do not do this, the old version might override the new one, depending on the order that RVISS encounters them.

See also

- *Tracer* on page 2-8
- *RVISS cycle types* on page 2-16
- *Pagetable module* on page 2-21
- *Default memory model* on page 2-28
- *Memory modeling with mapfiles* on page 2-29
- *Semihosting* on page 2-33
- *Peripheral models* on page 2-34
- *RVISS configuration files* on page 4-99
- Chapter 3 *Writing RVISS Models*
- *RealView Debugger Target Configuration Guide*.

2.4 Tracer

You can use Tracer to trace instructions, memory accesses, and events. The RVISS peripherals configuration file controls what is traced (see *RVISS configuration files* on page 4-99):

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The source code for Tracer is supplied in the following directory:

RVARMulator\ExtensionKit\...\...*platform*\

2.4.1 RealView Debugger support for tracing

You can trace RVISS models in RealView Debugger in the following ways:

- Configure and enable the RVISS Tracer feature.

  Note

  If you save the Tracer output to a file, you cannot use the RealView Debugger analysis features to analyze the captured trace.

- Use the trace and analysis features provided by RealView Debugger.

See also

- *Enabling the RVISS Tracer feature in RealView Debugger*
- *Configuring Tracer* on page 2-9
- *Interpreting trace file output* on page 2-11
- *RealView Debugger Trace User Guide*.

2.4.2 Enabling the RVISS Tracer feature in RealView Debugger

To enable the RVISS Tracer feature in RealView Debugger you must set bit 4 of the RVISS logging level variable `@rviss_log`. To do this, enter the following command at the RealView Debugger CLI prompt:

```
cexpression rviss_log=0x10
```

Note

Although you enable and disable the RVISS Tracer feature in RealView Debugger, what you trace is controlled by the `.ami` files.

See also

- *Configuring RVISS* on page 2-6.

2.4.3 Configuring Tracer

Tracer has its own section in the RVISS peripherals configuration file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The section has the following items:

```
{ Default_Tracer=Tracer
;; Output options - can be plaintext to file, binary to file or to RDI log
;; window. (Checked in the order RDILog, File, BinFile.)
;VERBOSE=True
;RDILog=True
RDILog=False
File=armul.trc
BinFile=armul.trc
;; Tracer options - what to trace
Architectural=False
TraceInstructions=True
TraceRegisters=True
OpcodeFetch=True
;;Normally True is useful, but sometimes it's too expensive.
TraceMemory=True
;TraceMemory=False
;Normally False, only True for timing tests.
;NB ARM10+XScale models do not emit ICycles on IBus or DBus.
;TraceIdle=True
TraceNonAccounted=False
;+TraceEvents=False
TraceEvents=True
EventMask=0
;;If there is a non-core bus, do we trace it (as well).
TraceBus=True
;; Flags - disassemble instructions; start up with tracing enabled;
Disassemble=True
;Set to True to output instuctions and memory-accesses in "ARMEIS" format.
TraceEIS=False
; Curently, precedes each access or opcode line with "T <corecycles>",
; or blank if the time has not changed.
TimeStamp=True

{Trace_EventNumbers
MMUEvent_DTLBWalk=True
MMUEvent_ITLBWalk=True
}
```

```
{TraceGenericNotifications
;We define generic notifications to have case-insensitive names.
;Generic notifications are registerred during instantiation by anything
; that can drive them.
;The data-format of each notification is negotiated at "route-links" time.
;This speeds up listeners, at the expense of complexity.
DTLB_LOCKDOWN=True
ITLB_LOCKDOWN=True
}

;TraceCoproRegisters=P11c0c1c2c3c4c5c6c7c8c9c10c11c12c13c14c15

StartOn=False
}
```

where:

`RDILog` Make sure this is set to **False**.

`File` Specifies the file where the trace information is written as a text file.

If you do not specify a path, the file is stored in:

install_directory`\RVD\Core\...\...\...\bin`

The other options control what is being traced:

`TraceInstructions` Traces instructions.

`TraceRegisters` Traces registers.

`OpcodeFetch` Traces instruction fetch memory accesses.

`TraceMemory` Traces memory accesses.

`TraceIdle` Traces idle cycles.

`TraceNonAccounted` Traces unaccounted accesses to memory. That is, those accesses made by RealView Debugger.

`TraceEvents` Traces events.

`TraceBus` Determines the following:

- `TRUE` Bus (off-chip accesses traced)
- `FALSE` Core (off-chip accesses not traced).

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

`TraceEIS` If set `TRUE`, changes output to a format compatible with other simulators. This enables tools to compare traces.

`StartOn` Instructs RVISS to trace as soon as execution begins.

Other tracing controls

You can also control tracing using:

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

Tracing is carried out only within the specified address range.

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

Tracing events

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

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

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

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

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

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

See also

- *Tracing events*
- *Events* on page 4-64.

2.4.4 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: Thu Feb 01 16:41:36 2007
Source: Armul
Options: Trace Instructions  (Disassemble)  Trace Memory Cycles
```

```
BNR4O___ A0000000 00000C1E
BNR8O___ 00008000 E28F8090 E898000F
BSR8O___ 00008008 E0800008 E0811008
BSR8O___ 00008010 E0822008 E0833008
BSR8O___ 00008018 E240B001 E242C001
MNR4O___ 00008000 E28F8090
IT 00008000 e28f8090 ADD      r8,pc,#0x90 ; #0x8098
MNR4O___ 00008004 E898000F
IT 00008004 e898000f LDMIA    r8,{r0-r3}
BNR4O___ A0000000 00000C1E
BNR8O___ 00008098 00007804 00007828
BSR8O___ 00008080 10844009 E3C44003
BSR8O___ 00008088 E2555004 24847004
BSR8O___ 00008090 8AFFFFFC EAFFFFF2
MNR8____ 00008098 00007804 00007828
BNR8O___ 000080A0 00007828 00007840
BSR8O___ 000080A8 E3A00840 E1A0F00E
BSR8O___ 000080B0 E92D400C E28F0014
BSR8O___ 000080B8 E5901000 E5900004
MNR8____ 000080A0 00007828 00007840
MNR4O___ 00008008 E0800008
IT 00008008 e0800008 ADD      r0,r0,r8
MNR4O___ 0000800C E0811008
IT 0000800C e0811008 ADD      r1,r1,r8
MNR4O___ 00008010 E0822008
```

Trace memory (M lines)

Memory (M) lines indicate:

- memory accesses, for cores without on-chip memory
- on-chip memory accesses, for cores with on-chip memory.

They 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 the access type:

- `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 (SWP instruction).
S	Indicates a speculative instruction fetch.
D	Indicates that the **DMORE** signal of the ARM9TDMI® data interface is HIGH.
<address>	The address in hexadecimal format, for example 00008008.
<data>	Indicates One of the following: *value* gives the read/written value, for example EB00000C (wait) indicates **nWAIT** was LOW to insert a wait state (abort) indicates **ABORT** was HIGH to abort the access.

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

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

Trace instructions (I lines)

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

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

For example:

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

where:

IT	Indicates that the instruction was taken.
IS	Indicates that the instruction was skipped (almost all ARM instructions are conditional).

<instr_addr>	The address of the instruction in hexadecimal format, for example 00008044.
<opcode>	The opcode in hexadecimal format, for example e04ec00f.
<disassembly>	The disassembly (uppercase if the instruction is taken), for example, SUB r12,r14,pc. This is optional and is enabled by setting Disassemble=True in peripherals.ami.

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

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

Trace 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>	The first of a pair of words, such as the PC value.
<word2>	The second of a pair of words, such as the aborting address.
<event_number>	An event number, for example 0x10005. This is the MMU Event_ITLBWalk.

Trace registers (R lines)

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

```
R <register>=<newvalue>[,<anotherregister>=<newvalue>[...]]
```

For example:

```
R r14=20000060, cpsr=200000d3
```

where:

<register>	A register that has a new value as a result of the current instruction.
<newvalue>	The new contents of <register>.

Trace bus (B lines)

The format of bus (B) lines is the same as the format of M lines. B lines indicate off-chip memory accesses.

See also

- *Tracing events* on page 2-11
- *Events* on page 4-64.

2.5 RVISS cycle types

In addition to simulating instruction execution on ARM cores, RVISS counts bus and processor cycles.

This section describes the meaning of the various types of cycles counted.

Note

RVISS does not model the AMBA *Advanced eXtensible Interface* (AXI) because the bus modeling is not accurate enough to differentiate between *Advanced High-performance Bus* (AHB) and AXI accesses.

2.5.1 Uncached von Neumann cores

Table 2-1 shows the meanings of cycle types for uncached von Neumann cores, such as the ARM7TDMI®.

Table 2-1 Cycle type meanings for uncached von Neumann cores

Cycle type	SEQ signal	nMREQ signal	Meaning
S_Cycles	1	1	Sequential cycles. See *Sequential cycles* for details.
N_Cycles	0	1	Nonsequential cycles. The CPU requests a transfer to or from an address unrelated to the address used in the immediately preceding cycle.
I_Cycles	1	0	Internal cycles. The CPU does not require a transfer because it is performing an internal function.
C_Cycles	0	0	Coprocessor cycles.
Total	-	-	The sum of S_Cycles, N_Cycles, I_Cycles, C_Cycles, and Waits.
IS	-	-	Merged I-S cycle. See *Merged I-S cycles* on page 2-17 for details.

Sequential cycles

The CPU requests transfer to or from:

- the same address as the address accessed in the immediately preceding cycle
- an address that is one word after the address accessed in the immediately preceding cycle

- for Thumb instruction fetches only, an address that is one half-word after the address accessed in the immediately preceding cycle.

Merged I-S cycles

A memory controller can start speculatively decoding an address during an I-Cycle. If the I_Cycle is followed by an S_Cycle, the memory controller can be ready to issue it earlier than otherwise. The timing of this cycle depends on the memory controller implementation.

2.5.2 Uncached Harvard cores

Table 2-2 shows the meanings of cycle types for uncached Harvard cores, such as the ARM9TDMI.

Table 2-2 Cycle type meanings for uncached Harvard cores

Cycle types	Instruction bus	Data bus	Meaning
Core cycles	-	-	The total number of ticks of the core clock. This includes pipeline stalls because of interlocks and instructions that take more than one cycle.
ID_Cycles	Active	Active	-
I_Cycles	Active	Idle	-
Idle Cycles	Idle	Idle	-
D_Cycles	Idle	Active	-
Total	-	-	The sum of core cycles, ID_Cycles, I_Cycles, Idle_Cycles, D_Cycles, and Waits.

2.5.3 Cached cores with MMUs or MPUs and AMBA ASB interfaces

Table 2-3 shows the meanings of the bus cycle types for cached cores with AMBA™ *Advanced System Bus* (ASB) interfaces.

ARM920T™ is an example of a cached core with a *Memory Management Unit* (MMU). ARM940T™ is an example of a cached core with a *Memory Protection Unit* (MPU).

Table 2-3 Cycle type meanings for cached cores with AMBA ASB interfaces

Cycle types	Meaning
A_Cycles	An address is published speculatively. No data is transferred. Listed as I_Cycles in RealView Debugger `@stats_`*`symbolname`* symbols.
S_Cycles	Sequential data is transferred from the current address.

There are no N_Cycles for these cores. Nonsequential accesses use an A_Cycle followed by an S_Cycle. This is the same as a merged I-S cycle.

See also

- *Internal cycle types for cached cores* on page 2-19.

2.5.4 Cached cores with MMUs or MPUs and AMBA AHB interfaces

Table 2-4 shows the types of transfer that can occur on the AHB. ARM946E-S™, for example, is a cached core with an AHB interface.

Table 2-4 Cycle types on AMBA AHB interfaces

Cycle types	Meaning
IDLE	The bus master does not want to use the bus. Slaves must respond with a zero wait state **OKAY** response on **HRESP**.
BUSY	The bus master is in the middle of a burst, but cannot proceed to the next sequential access. Slaves must respond with a zero wait state **OKAY** response on **HRESP**.
NON-SEQ	The start of a burst or single access. The address is unrelated to the address of the previous access.
SEQ	Continuing with a burst. The address is equal to the previous address plus the data size.

See also

- *Internal cycle types for cached cores*.

2.5.5 Internal cycle types for cached cores

Table 2-5 shows the meaning of internal cycle types for cached cores.

Table 2-5 Internal cycle types for cached cores

Cycle types	Meaning
F_Cycles	Fast clock (**FCLK**) cycles. These are internal core cycles accessing the cache. F_Cycles is not incremented for uncached accesses because the core clock switches to the bus clock.
Core Cycles	Core cycles are clock ticks to the core. Core Cycles are incremented for each tick, whether the core is running **FCLK** (cache accesses) or bus clock (**BCLK**, non-cache accesses).
True Idle Cycles	Idle cycles that are not part of a merged I-S cycle.

Note

If you want to count execution time, use external bus cycle counts. You cannot use F_Cycles to count execution time, because F_Cycles does not increment for uncached accesses.

2.5.6 StrongARM1

Table 2-6 shows the meaning of cycle types reported for StrongARM1.

Table 2-6 StrongARM specific cycle types

Cycle types	Meaning
Core_Idle	No instruction fetched from instruction cache. No data fetched from data cache.
Core_IOnly	Instruction fetched from instruction cache. No data fetched from data cache.
Core_DOnly	No instruction fetched from instruction cache. Data fetched from data cache.
Core_ID	Instruction fetched from instruction cache. Data fetched from data cache.

2.5.7 Core-specific verbose statistics

There is a line in the `default.ami` file:

```
Counters=False
```

You can change this to read:

```
Counters=True
```

If you do this, additional statistics, such as cache hits and cache misses, are counted by RVISS and appear in RealView Debugger as `@stats_`*`symbolname`* symbols. These statistics are core-specific.

2.5.8 See also

- *RealView Debugger User Guide*
- *RealView Debugger Target Configuration Guide*
- *RealView Debugger Command Line Reference Guide*.

2.6 Pagetable module

This section describes the RVISS pagetable module.

2.6.1 Overview of the pagetable module

The pagetable module enables you to run code on a model of a system with an MMU or an MPU, without having to write initialization code for the MMU or MPU.

Note

This module enables you to debug code, or perform approximate benchmarking. For a real system. you must write initialization code to set up the MMU or MPU. You can debug your initialization code on RVISS by disabling the pagetable module.

On models of *ARM architecture v4* (ARMv4), ARMv5, and ARMv6 processors with an MMU, the pagetable module sets up pagetables and initializes the MMU. On processors with an MPU, the pagetable module sets up the MPU. You can control whether to include the pagetable model in the following ways:

- Use the **Default Page-Tables** checkbox in the RealView Debugger ARMulator Configuration dialog box to set `PAGETAB` to `No_Pagetables` (the default) or `Default_Pagetables`. RealView Debugger sets the `PAGETAB` key in the root of the configuration.
- Find the `PAGETAB` variable in the RVISS configuration file:

  `install_directory``\RVARMulator\ARMulator\...\...\`*`platform`*`\default.ami`

  Alter the entry as appropriate (see also `Pagetables=$PAGETAB` in this file):

  ```
  {PAGETAB=Default_Pagetables
  }
  {PAGETAB=No_Pagetables
  }
  ```

- Edit the Pagetables section in the file:

  `install_directory``\RVARMulator\ARMulator\...\...\`*`platform`*`\peripherals.ami`

  This controls the contents of the pagetables, and the configuration of the caches and MMU or MPU. To locate the `Pagetables` section, find the line:

  ```
  {Default_Pagetables=PageTables
  ```

  Use this option only if you want to override the RealView Debugger configuration.

See also

- *ARM Architecture Reference Manual*
- the Technical Reference Manual for the processor you are simulating.

2.6.2 Controlling the MMU, MPU or MPU and cache

The first set of flags enables or disables features of the caches and MMU, MPU, or ARMv6 Memory Protection Unit (*MPU*). For the MMU and MPU, the flags are:

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

For the MPU used by ARM1156T2-S and ARM1156T2F-S processors, the flags are:

```
PhysicalBase=0
Size=4GB
Cacheable=Yes
Shareable=No
Bufferable=Yes
AccessPermissions=3
TEX=0
Execute_Never=0
```

Each flag corresponds to a bit in the system control register, c1 of CP15.

Some flags only apply to certain processors. For example:

- `BranchPredict` only applies to the ARM810
- `ICache` applies to StrongARM®-110 and ARM940T processors, but not ARM720T™ for example.

These flags are ignored by other processor models.

Note

See the RealView Debugger documentation for supported processors.

The FastBus flag is used by some cores such as ARM940T:

- If your system uses FastBus Mode, set FastBus=Yes for benchmarking.
- If you set FastBus=No, RVISS assumes that the memory clock is slower than the core clock by a factor of the memory clock configuration value MCCfg. RVISS does not model Asynchronous mode.

Note

ARM925T only uses the same clock for both core and bus interface. MCCfg therefore has no effect on an ARM925T.

The MMU flag is used to enable the MPU in processors with an MPU.

See also

- *Application Note 51 ARMulator Cache Models*
- *Application Note 93 Benchmarking with ARMulator*
- the Technical Reference Manual for your core.

2.6.3 Controlling registers 2 and 3

The following options apply only to processors with an MMU:

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

They control:

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

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

2.6.4 Memory regions

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

By default, a description of a two regions is contained in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The regions have the following items:

```
{ Region[0]
VirtualBase=0
PhysicalBase=0
Size=4GB
Cacheable=No
Bufferable=No
Updateable=Yes
Domain=0
AccessPermissions=3
Translate=Yes
}

{ Region[1]
VirtualBase=0
PhysicalBase=0
Size=128Mb
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 region *n*, starting with `Region[0]`. *n* is an integer.

`VirtualBase` This 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` 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 an MPU it must be aligned to a boundary that is a multiple of the size of the region.

`Size` The size of this region. On a processor with an MMU `Size` must be a whole number of megabytes. On a processor with an MPU, `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 are cached.

`Bufferable` Specifies whether the region is to be marked as bufferable. If it is, writes to the region use the write buffer.

`Updateable`	This applies only to the ARM610 processor. It controls the `U` bit in the translation table entry.
`Domain`	This applies only to processors with an MMU. It specifies the domain field of the table entry.
`AccessPermissions`	Specifies the access controls to the region. See the processor technical reference manual for more 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.

You must ensure that you do not define more regions than your target hardware supports. At least one region must be defined.

2.6.5 Pagetable module and memory management units

Processors such as ARM720T™ 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 the file:

`install_directory``\RVARMulator\ARMulator\...\...\`*`platform`*`\peripherals.ami`

For example, the default configuration details given in *Memory regions* on page 2-23 define the following page table:

- The entire address space, 4GB, is defined as a single region. This region is not cacheable or bufferable. Virtual addresses are mapped directly to the same physical addresses over the whole address space.
- The first 128MB of the address space is defined as a second region overlapping the first. This region is cacheable and bufferable. Virtual addresses are mapped directly to physical addresses.

They also set up the control registers as follows:

- 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.

2.6.6 Pagetable module and memory protection units

Processors such as ARM740T and ARM940T have an MPU.

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

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 MPU

For an ARM740T, the MPU 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. This is configured in the `AccessPermissions` line.
- 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 technical reference manual for the processor for more details.
- The protection region base and size register for region 1 is initialized to set the size of region 0 as 128MB and enabled.

ARM940T MPU

For an ARM940T, the MPU is initialized as follows:

- The P, D, W, and I bits are set in the configuration register, register 1, to enable the MPU, 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 RealView 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 RealView 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 regions 0 and 1 are initialized to mark the sizes of the regions and mark them as enabled. The protection region base and size registers for all regions are 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 technical reference manual for the processor for more details.
- Register 7 is a control register. Reading from it is unpredictable. At startup RealView 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 RealView 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 RVISS. These control whether the cache replacement algorithm is random or round-robin.

2.7 Default memory model

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

If you do not specify a mapfile, RVISS uses the default memory model. The default memory model is not visible in RealView Debugger.

Note

If you specify a mapfile, the mapfile does not replace the default memory model, but sits between the memory model and the core, and inserts events, for example wait states and aborts. The memory map defined in the map file is visible in the **Mapfile** tab of the RealView Debugger Registers pane.

The default memory model routes memory accesses to memory-mapped peripheral models as appropriate. Routing is based on configuration details you provide in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

or in another `.ami` file.

2.8 Memory modeling with mapfiles

This section describes RVISS memory modeling using map files.

2.8.1 Overview of memory modeling with mapfiles

`mapfile` 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.

RVISS simulates each memory access as it occurs. It counts wait states according to the type of memory access.

The RealView Debugger internal variables `@mapfile_`*`symbolname`* and `@stats_`*`symbolname`* give details of accesses of each cycle type, regions of memory accessed and time spent accessing each region.

See also

- *Map files* on page 4-95
- *RealView Debugger User Guide*
- *RealView Debugger Target Configuration Guide*
- *RealView Debugger Command Line Reference Guide*.

2.8.2 Clock frequency

You can configure the clock frequency used by `mapfile` from the RealView Debugger ARMulator Configuration dialog box.

The clock frequency is used to determine the number of wait states to be added to each memory access, and to calculate time from number of cycles.

If you do not specify a clock speed, a value of 20MHz is used. If you specify a number without units, the units are Hz. You can specify Hz, kHz, or MHz.

See also

- *RealView Debugger Target Configuration Guide*.

2.8.3 Selecting the mapfile memory model

You specify a memory map file in the `peripherals.ami` configuration file, or your own variant. RealView Debugger automatically applies the map memory model defined in the file when you connect to the RVISS model.

See also

- *Configuring the map memory model* on page 2-31.

2.8.4 How the mapfile memory model calculates wait states

For nonsequential/sequential reads/writes to various regions of memory, the memory map file specifies access times in nanoseconds. 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. Consider this when designing your system.

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.

If the access time is 60ns (only 14% faster) the model inserts 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

For accurate simulation of the real hardware, access times specified in the memory map file must include propagation delays and memory controller decode time in addition to the access time of the memory devices. For example, for 70ns RAM, if there is a 10ns propagation delay, configure the map file as 80ns.

2.8.5 Configuring the map memory model

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

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The entry has the following items:

```
{ Default_Mapfile=Mapfile
AMBABusCounts=False
;SpotISCyles=True|False
SpotISCyles=True
;ISTiming=Late|Early|Speculative
ISTiming=Late
;MAPFILETOLOAD=<name>
}
```

Specifying a mapfile to load

You can specify a mapfile to load by replacing `<name>` in the `MAPFILETOLOAD` line with the path and filename of the mapfile to load, and removing the commenting characters from the line.

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 the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

Merged I-S cycles

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

You can disable this by setting `SpotISCycles=False` in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

However, this is likely to result in exaggerated performance figures, particularly when modeling cached ARM processors.

The model can simulate merged I-S cycles using one of the following 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 fewer wait state.

Early This starts the decode when the ARM processor 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 states. (Whether or not there are fewer wait states depends on the cycle time and the nonsequential access time for that region of memory.)

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

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

See *RVISS cycle types* on page 2-16 for details of merged I-S cycles.

2.9 Semihosting

Semihosting provides code running on an ARM target use of facilities on a host computer that is running an RealView Debugger. Examples of such facilities include the keyboard input, screen output, and disk I/O.

See *RealView Compilation Tools Libraries and Floating Point Support Guide* for more details.

2.9.1 Semihosting configuration

The semihosting SVC handler configuration is controlled by a section in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The section has the following items:

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

Note

RealView Debugger overrides the `AngelSVCARM` and `AngelSVCThumb` settings.

2.10 Peripheral models

RVISS includes several peripheral models. This section gives basic user information about them.

Note

The Stackuse model is not supported.

2.10.1 Configuring RVISS to use the peripheral models

Enable or disable each peripheral model by changing the relevant entry in your copy of the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\default.ami

For example:

```
{ WatchDog=No_watchdog
}
```

can be changed to:

```
{ Watchdog=Default_WatchDog
}
```

Other peripheral models are controlled in the same way, using the `No_` and `Default_` prefixes to the peripheral names.

2.10.2 Configuring details of the peripherals

Configuration details for the peripheral models are in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami.

See also

- *Configuring RVISS* on page 2-6.

2.10.3 Interrupt controller

The interrupt controller is an implementation of the reference interrupt controller.

The configuration of the interrupt controller model is controlled by a section in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The section has the following items:

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

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

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

See also

- *Interrupt controller* on page 4-111.

2.10.4 Timer

The timer is an implementation of the reference timer. It provides two counter-timers.

The configuration of the timer model is controlled by a section in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The section has the following items:

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

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

`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.

See also

- *Timer* on page 4-113.

2.10.5 Watchdog

Use Watchdog to prevent a failure in your program locking up your system. If your program fails to access Watchdog before a predetermined time, Watchdog halts RVISS and returns control to RealView Debugger.

Note

This is a generic model of a watchdog timer. It is supplied to help users model their system environment. It does not model any actual hardware supplied by ARM Limited.

The Watchdog configuration is controlled by a section in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The section has the following items:

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

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

This is a two-timer watchdog.

If `StartOnReset` is `True`, the first timer starts on reset. If `StartOnReset` is `False`, the first timer starts only when your program writes the configured key value to the `KeyValue` register. This is located at the address given in the `Range:Base` line (`0xB0000000`).

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 RVISS if the second timer times out. You can continue to execute, or debug.

If RunAfterBark is False, Watchdog halts RVISS and returns control to RealView Debugger.

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.

When the Watchdog activates, execution stops and the following messages are displayed in the **StdIO** tab of the RealView Debugger Output pane:

```
Fatal System Error: Watchdog timed out at cycle_count!!
Halting System to allow debug.
Watchdog has halted emulation and reset
the target.
```

After the Watchdog activates:

1. Reset the target to reset the timers.
2. Reload your image.

See also

- *RealView Debugger User Guide*.

2.10.6 Tube

The tube is a memory-mapped 4-byte register. If you write a printable character to each byte, the characters appear in the **StdIO** tab of the RealView Debugger Output pane. It enables you to check that writes are taking place to a specified location in memory.

The address at which the Tube is mapped is controlled by an entry in the file:

install_directory\RVARMulator\ARMulator\...\...*platform*\peripherals.ami

The entry has the following items:

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

This is the default address.

Considerations when using tube

Be aware of the following:

- Semihosting must be enabled for the RVISS target.
- It is suggested that you also use Watchdog to prevent a failure in your program locking up your system. You might want to set `RunAfterBark=False` to return control back to RealView Debugger if, for example, your program gets into an endless loop when writing to the memory mapped register.
- Tube intercepts writes to the specified address. Therefore, the characters do not appear in the specified location in RealView Debugger Memory pane.
- Do not use the RVISS map file feature. If you do, then no characters are output.

See also

- *Watchdog* on page 2-36
- *RealView Debugger User Guide*.

Chapter 3
Writing RVISS Models

This chapter is intended to assist you in writing your own models to add to *RealView® ARMulator® Instruction Set Simulator* (RVISS). It contains the following sections:

3.1 The RVISS extension kit

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

3.1.1 Location of files

The RVISS extension kit contains the source code of some models. You can make copies of these models, and modify the copies.

Location of source files

The source code of the models for you to copy is supplied in:

install_directory\RVARMulator\ExtensionKit\...\...*platform*\armulext

In this path:

- *platform* is:
  - `win_32-pentium` for Windows
  - `linux-pentium` for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

Location of header files

Header files are supplied in:

install_directory\RVARMulator\ExtensionKit\...\...*platform*\armulif

In this path:

- *platform* is:
  - `win_32-pentium` for Windows
  - `linux-pentium` for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

Location of makefiles

Makefiles are supplied in:

install_directory\RVARMulator\ExtensionKit\...\...*platform*\armulext*model*.b*target*

In this path:

- *platform* is:
  - win_32-pentium for Windows
  - linux-pentium for Red Hat Linux.
- *model* is the model name. For example, millisec.b is the directory for the peripheral model millisec that implements a simple millisecond timer counter.
- *target* is:
  - intelrel for Windows
  - linux86 for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

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).

Example code is supplied in:

$ARMROOT\RVARMulator\ExtensionKit\...\...*platform*\armulext\

The libraries that this example code links against are supplied in one of:

- $ARMROOT\RVARMulator\ExtensionKit\...\...*platform*\armulif\armulif.b*platform*\
- $ARMROOT\RVARMulator\ExtensionKit\...\...*platform*\clx\clx.b*platform*\
- $ARMROOT\RVARMulator\ExtensionKit\...\...*platform*\rdi\rditools.b*platform*\

You must ensure that the makefile contains the correct build options for linking against RVISS.

3.1.2 Supplied models

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

- basic models
- peripheral models.

Basic models

`tracer.c` The tracer module can trace instruction execution and events from within RVISS (see *Tracer* on page 4-93). You can link your own tracing code onto the tracer module.

`profiler.c` Not supported by RealView Debugger. However, you can use RealView Debugger tracing to capture profiling information.

See the *RealView Debugger Trace User Guide* for more details.

`pagetab.c` On reset, this module sets up cache, MPU or MMU and associated pagetables inside RVISS (see *Pagetable module* on page 2-21).

`nothing.c` This model does nothing. You can use this in the `peripherals.ami` file to disable models (see *Configuring RVISS to disable a model* on page 3-13).

`semihost.c` This model provides the semihosting SVCs described in *RealView Compilation Tools Libraries and Floating Point Support Guide*.

`dcc.c` This is a model of a *Debug Communications Channel* (DCC).

`mapfile.c` This model enables you to specify the characteristics of a memory system. See *Map files* on page 4-95 for more information.

`flatmem.c` `flatmem` models a zero-wait state memory system. See *Default memory model* on page 2-28 for more information.

`validate.c` `validate` is a coprocessor model used for validation with some cores. It can generate delayed IRQ and FIQ signals, for example.

Peripheral models

`intc.c` See *Interrupt controller* on page 2-34. `intc` is a model of the interrupt controller peripheral described in the *Reference Peripherals Specification* (RPS).

`timer.c` See *Timer* on page 2-35. `timer` is a model of the RPS timer peripheral. Two timers are provided. `timer` must be used in conjunction with an interrupt controller, but not necessarily `intc`.

`millisec.c`	A simple millisecond timer.
`watchdog.c`	Watchdog. See *Watchdog* on page 2-36. `watchdog` is a generic watchdog model. It does not model any specific watchdog hardware, but provides generic watchdog functions.
`tube.c`	Tube. See *Tube* on page 2-37. `tube` is a simple debugging aid. It enables you to check that writes are taking place to a specified location in memory.

3.2 Writing a new peripheral model

This section describes how to write a new peripheral model.

3.2.1 Using a sample model as a template

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

1. Select which model is closest to the model you want to write. This might be, for example, Timer.
2. Copy the source file, in this case `timer.c`, with a new name such as *`mymodel`*`.c`.
3. Copy the make subdirectory, in this case `timer.b`, with a corresponding new name, in this case *`mymodel`*`.b`.
4. Find the `Makefile` for your model (see *Location of files* on page 3-2).

   Load `Makefile` into a text editor and change all instances of `timer` to *`mymodel`*.

You can now edit *`MyModel`*.

Note

Although some of the ARM11 hardware processors are AXI, the bus modeling is not accurate enough to differentiate between AHB and AXI accesses. Therefore, you must use the `ARMul_MemType_ARMissAHB` memory type when building peripheral models for all ARM11 processor models (ARM1136, ARM1156, ARM1176, and MPCore™).

3.2.2 Return values

A model must return one of the following states for memory accesses:

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

3.2.3 Initialization, finalization, and state macros

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

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

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

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

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

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

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

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

The data type is `your_modelState`.

3.2.4 Registering your model

You can register your model in the following ways:

- If you are writing a peripheral model that is designed to hang off a bus-decoder, the model should call `registerPeripFunc()`. This enables RVISS to call your model with accesses to memory locations that belong to your model. The bus-decoder is generally implemented by the end-of-chain memory, `Flatmem`, and ensures that the peripheral will be called for the range of addresses that it is registered for. An example of this is shown in the `millisec.c` file that is supplied with the examples See *ARMul_BusRegisterPeripFunc* on page 4-79.
- If you are writing an end-of-chain memory you must do one of the following:
  - use `ARMulif_InsertMemInterface()` at a point just above `Flatmem` to intercept all calls to `Flatmem` and don't pass them on
  - change your configuration to load your own `.dll` file instead of `Flatmem`.
- If you are implementing a new tracer, then this usually inserts itself between the core and cache to intercept all L1 memory accesses. The source code for the tracer is supplied with the examples so that you can see how it does this.

3.3 Building a new model

RVISS finds its configuration files by looking for the `.dll`, `.so` or `.sdi` files along the set of paths in the `ARMCONF` environment variable. These paths are automatically set up by the installer. RVISS usually expects to find models in the executable path:

install_directory`\RVARMulator\ARMulator\...\...\`*platform*

When you build a new model, you must do one of the following:

- place it along the path set up by the installer. This is usually `$ARMROOT\RVARMulator\ARMulator\...\...\`*platform*
- append or prepend the directory containing the library (not the name of the library) to `ARMDLL`.

In paths referred to in this chapter:

- *platform* is:
  - `win_32-pentium` for Windows
  - `linux-pentium` for Red Hat Linux.
- *mymodel* is the name you have given the new model.
- *target* is:
  - `intelrel` for Windows
  - `linux86` for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

RVISS uses the first `.dll` that it finds.

3.3.1 How to build a new model

To build your new model:

1. Change your current directory to the `build_path` for your system.
2. Build the model using the make utility installed on your system. This might be one of:
   - `nmake` for Windows
   - `make` or `GNU make` for Red Hat Linux.

   The file is placed in the directory that contains the makefile.

   Note

   For clarity, the executable and build paths are referred to in this section as `exec_path` and `build_path`. See *Building a new model* on page 3-9 for the full path name.
3. Move the built file from the `build_path` to the `exec_path`. Depending on your system, the filename is:
   - on Windows, *`mymodel`*`.dll`
   - on Red Hat Linux, *`mymodel`*`.so`.

Note

If your makefile invokes Microsoft Visual Studio, you must use Version 6 (not an earlier or a later version). You must also recompile any existing plugins with this version.

Example code is supplied in:

`$ARMROOT\RVARMulator\ExtensionKit\...\...\`*`platform`*`\armulext\`

3.4 Configuring RVISS to use a new model

RVISS determines which models to use by reading the `.ami` and `.dsc` configuration files. See *RVISS configuration files* on page 4-99.

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

3.4.1 Adding a .dsc file

Create a file called *MyModel*`.dsc` and place it in the `ARMCONF` environment variable.

install_directory`\RVARMulator\ARMulator\...\...\`*platform*

MyModel`.dsc` must contain the following:

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

You do not need to supply a file extension for the `MyModel.dsc` file in `ToolConf`. Naming the filename without an extension ensures that your model works on all platforms. If you want to supply an extension, the file will be called either `MyModel.dll` or `MyModel.so`, depending on your system. see *How to build a new model* on page 3-10):

- *MyModel*`.dll`
- *MyModel*`.so`.

Either of these can be renamed to *MyModel*`.sdi`. Your build system then creates a file with the appropriate file extension.

`Nothing` is a predefined model that does nothing. The `No_`*MyModel*`=Nothing` line enables the use of `No_`*MyModel* in a `.ami` file. This enables a user to configure RVISS to exclude your model (see *Configuring RVISS to disable a model* on page 3-13).

You can include other configuration details in your *MyModel*`.dsc` file if required. For examples, see the supplied `.dsc` files in:

install_directory`\RVARMulator\ARMulator\...\...\`*platform*

Note

You can keep your model files in a separate directory if required. This enables you to use a network installation of RVISS without causing problems with other users who are using different models. This also ensures that your files still work if a newer version of RVISS is installed.

3.4.2 Editing default.ami and peripherals.ami

This description assumes that your model was based on Timer:

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

   ```
   {Timer=Default_Timer
   }
   ```

2. Add the reference to your model:

   ```
   {Timer=Default_Timer
   }

   {MyModel=Default_MyModel
   }
   ```

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

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

   ```
   { Default_Timer=Timer
   .
   .
   .
   }
   ```

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

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

3.5 Configuring RVISS to disable a model

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

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

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

2. Change them to read:

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

This uses the `nothing.c` model to override the `tube.c` model. `nothing` ignores any configuration details such as `Range:Base`.

Chapter 4
RVISS Reference

This chapter gives reference information about *RealView® ARMulator® Instruction Set Simulator* (RVISS). It contains the following sections:

4.1 SimRdi_Manager interface

RealView Debugger enables you to connect to RVISS models using a RealView Connection Broker interface. An intermediate interface is required to interface the various RVISS modules and RealView Connection Broker. This intermediate interface is called SimRdi_Manager. SimRdi_Manager does not enable RVISS models to manipulate the RealView Connection Broker interface directly. Instead the interface presents several services that models can call or register against.

Note

Although RealView Debugger does not support the *Remote Debug Interface* (RDI) for hardware targets, this interface is still supported.

4.1.1 Using the SimRdi_Manager interface

During initialization an RVISS model must install a listener to wait for an announcement from SimRdi_Manager. When the listener is called the model must register with SimRdi_Manager.

When you create an RVISS model that communicates through RealView Connection Broker, you must decide on the services that your model is to provide. Your model must notify SimRdi_Manager which services it is providing.

See also

- *Connections to RVISS in RealView Debugger* on page 2-3
- *Supported SimRdi_Manager services* on page 4-5
- *Adding a SimRdi_Manager listener* on page 4-6
- *Advertising the SimRdi_Manager services provided by your model* on page 4-9
- *RealView Debugger Target Configuration Guide*.

4.1.2 Header files

The interfaces that models use to interface to SimRdi_Manager are kept in the following directory of the rebuild kit:

install_directory\RVARMulator\ExtensionKit\...\...*platform*\armulif

In this path:

- *platform* is:
  - win_32-pentium for Windows
  - linux-pentium for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

These interfaces are:

simrdi_registration_event.h

An interface for models to determine whether or not RVISS is being controlled by SimRdi_Manager.

uniregs_registration_event.h

If you are writing a replacement standard ARM® coprocessor without defining non-coprocessor registers, then include this file.

mini_simrdi_manager.h

The main header file that you must include. It provides anything else that is required.

tmgrem.h An internal header file. The following enumerations and unions in this file are useful (see *Stopping RVISS* on page 4-18 for details):

- REGVAL
- STATUS_INFO
- GEN_SIGNALS
- STATUS_MODE.

Note

Do not use any other enumerations, unions, macros, or structures in this file.

4.1.3 Supported SimRdi_Manager services

The services supported by the SimRdi_Manager are:

Global break conditions

These are the concept of a synchronous event that is not tied to a specific address. For example, the exceptions Reset and Undefined Instruction are already automatically registered as Global Breaks in SimRdi_Manager.

See *Global break service* on page 4-11 for details on how to advertise this service.

Adding registers

Models with registers that users can interrogate and potentially write to can expose those registers. Normally these symbols are named `@name`. You are strongly encouraged to expose the internal state of models in this way.

See *Register services* on page 4-12 for details on how to advertise this service.

Adding register window tabs

If you add registers, then depending on how you add them, they might automatically appear in the register window. Otherwise, if you want them to appear in the register window you must explicitly write a register window tab structure to do this and expose it to SimRdi_Manager.

See *Register windows service (regwin)* on page 4-16 for details on how to advertise this service.

Note

Currently, a model can use only these three SimRdi_Manager services. The remaining services are for future releases of RVISS, and are offered only as comments in the header files, with no support by ARM Limited.

4.1.4 Adding a SimRdi_Manager listener

For your RVISS model to communicate through the RealView Connection Broker, it must register with SimRdi_Manager, and advertise the services it can provide. See *Advertising the SimRdi_Manager services provided by your model* on page 4-9 for more information on how to advertise the services provided by your model.

To add a SimRdi_Manager listener:

1. Define a listener function, for example, `Model_SimRdi_Listener`.
2. Register the listener function using `ARMulif_InstallSimRdiRegistration()`.

The prototype for the `ARMulif_InstallSimRdiRegistration()` function is in the header file `simrdi_registration_event.h`. This file is automatically included if you include the header file `mini_simrdi_manager.h`. See *ARMulif_InstallSimRdiRegistration* on page 4-8 for more details.

The listener is passed a structure of type `SimRdiRegistrationProcVec` and the handle that was passed into `ARMulif_InstallSimRdiRegistration()`.

`SimRdiRegistrationProcVec` contains:

- a signature that SimRdi_Manager checks for you
- a toolconf that you can use as required
- pointers to structures containing methods.

The number of these last elements is also in this structure so that more can be added later and the model can check that a particular pointer it requires is there.

In this release, only two members are defined:

`SimRdiProcVec* simrdiprocvec;`
: This is the interface to the SimRdi_Manager.

`UniregsRegistration* uniregsprocvec;`
: An abbreviated interface that is simpler to use for those models that are implementing or re-implementing the standard ARM coprocessors.

Example 4-1 on page 4-7 shows a skeleton SimRdi_Manager listener.

Example 4-1 Skeleton SimRdi_Manager listener

```
void Model_SimRdi_Listener(SimRdiRegistrationProcVec *registration,void *handle)
{
    /*  typically a model must first check that the service it wants
        is in the registration. */
    if (registration->number < 1)
        return;                          /* no services available! */
    else
    /*  registration->simrdiprocvec is an interface to
        SimRdi_Manager, typically a model would now save this
        pointer in its state, which people usually choose to
        pass in as handle.
        simrdiprocvec will *not* move in memory so it is safe to
        store it. */
    {
        Model_State *state = (Model_State*)handle;
        state->srpv = registration->simrdiprocvec;
        ...;        /*  go on to register services it can provide against
                        SimRdi_Manager using the methods in simrdiprocvec */
    }
}
```

The structure `SimRdiProcVec` is defined in the header file `mini_simrdi_manager.h`.

A model advertises to SimRdi_Manager the services it provides by requesting an advert, filling in the advert, and then advertising the advert. The `SimRdiProcVec` contains pointers to functions to do this.

For example, you might have a service called `service` and if `srpv` is a variable holding a pointer to a `SimRdiProcVec`, then `srpv->service` is a structure containing methods that enable you to manipulate the service.

ARMulif_InstallSimRdiRegistration

This prototype function installs an unknown RDIInfo handler that captures a specific RDIInfo call that has been defined for this purpose.

Syntax

```
void ARMulif_InstallSimRdiRegistration( RDI_ModuleDesc* coredesc,
SimRdiRegistrationListener* func,
void* handle_to_pass_to_listener);
```

where:

coredesc Used in multicore systems to describe the core it is connected to.

func The listener function, for example, Model_SimRdi_Listener. See *SimRdiRegistrationListener* for details.

handle_to_pass_to_listener

The handle of the unknown RDIInfo handler that is to be passed to the listener function.

SimRdiRegistrationListener

The prototype for a listener function that is registered with SimRdi_Manager using the ARMulif_InstallSimRdiRegistration function.

Syntax

```
typedef void SimRdiRegistrationListener(
                                SimRdiRegistrationProcVec* registration,
                                void* handle );
```

where:

registration The services available in the version of SimRdi_Manager that calls the listener function.

handle The handle specified in the ARMulif_InstallSimRdiRegistration function.

4.1.5 Version information

By examining the version information in `SimRdiProcVec->version->major` and `SimRdiProcVec->version->minor`, the model can determine what version of the `SimRdiProcVec` it is using, and if the service exists in that version:

- a major number revision means that the model is incompatible with this version of `SimRdiProcVec`
- a minor revision number greater than or equal to the one that the model knows about means that the service exists.

When a model is compatible with the `SimRdiProcVec` version, the service provided by the model exists. Therefore, the model can safely obtain the version number of the service using:

```
srpv->service->version;
```

To obtain the major version number use `srpv->service->version & 0x0000FF00`.

To obtain the minor version number use `srpv->service->version & 0x000000FF`.

Note

The top 16 bits are reserved.

The interpretation of the major and minor numbers of the service version are the same as that of `SimRdiProcVec`. That is, where the major revision number has changed, the service must not be used. However, a minor revision number greater than or equal to the one that the model knows about is compatible, and the service can be used safely.

4.1.6 Advertising the SimRdi_Manager services provided by your model

During the initialization, your RVISS model must notify SimRdi_Manager which services it is providing (see *Supported SimRdi_Manager services* on page 4-5). You do this by creating adverts.

To request an advert call `srpv->service->c_new(srpv->service)`. SimRdi_Manager populates the advert with default values. If a peripheral model was created with a lower version than your current SimRdi_Manager, then the peripheral model alters only those fields in the advert that it knows about. Therefore, SimRdi_Manager can determine the version of SimRdi_Manager that was used to build the peripheral model by checking the unaltered fields.

A typical sequence during model initialization is:

```
SimRdi_Service_Advert *ad = srpv->service->c_new(srpv->service);
state->saved_advert_id = ad->x.id;  /* save id of advert */
...;                                /* fill out advert */
srpv->service->advertise(srpv->service, ad);
ad = NULL;                      /* do not assume that ad is still valid */
```

Then during destruction of the model it calls:

```
if (advertised_a_service)
{
    srpv->service->destroy(srpv->service, state->saved_advert_id);
}
```

Each advert has a structure `x` that is of type `SimRdi_Advert_Base`. This type has standard fields that you can fill in:

`name` The name of the advert, useful for debugging purposes.

`id` A unique identifier for the advert that can be used to destroy the advert when the model exits. After the advert has been advertised, you must not assume that the pointer to the advert remains valid. SimRdi_Manager is free to move the advert to optimize access to it.

`handle` Data that is specific to the model. You can choose how you use this field.

`coredesc` If the model is attached to a single processor then the model can include a core description for the processor that instantiated it. This is required in multiprocessor systems to identify the core that the model belongs to.

Note

Multiprocessor models are not available in this release. However, filling in this field ensures that your model works with a multiprocessor RVISS when this feature is available.

`simrdi_manager_data`

Private data to SimRdi_Manager.

Note

You must not modify this.

`config_flags` The interpretation of these flags depends on the type of the advert. Typically some of the bits are used to sort the adverts for a particular service. Also a bitwise OR of all the adverts registered for a particular service is available in the `srpv->service` structure.

`notice` Is reserved for future expansion, and is set to `NULL`.

4.1.7 Global break service

A *global break service* represents the concept of a synchronous condition that cannot necessarily be assigned to an address. Examples are the processor exceptions, undefined instruction, and supervisor call.

Global breaks are advertised through the type `SimRdi_Global_Breaks_Advert`, which has the following members that the model must fill in:

`x` The base advert that the model populates.

`len` The number of strings in the `global_breaks` array.

`global_breaks`

A list of strings that describe the global break. These strings appear in the Processor Events dialog box.

`handle_for_function`

Passed as an argument to the `simrdi_global_breaks` function. The function must return either `SIM_REGISTER_ACCESS` on error or `SIM_OK` on success.

`simrdi_global_breaks`

A function that is called to turn global breaks on and off, and to determine the current state of the global breaks.

`start_global_number`

A pointer to an `int` that is filled in the global break number of the first global break in `global_breaks`. They are numbered consecutively from `*start_global_number`. The position pointed at must remain valid for the lifetime of the model.

When a global break happens, and it is enabled, then the model uses as the global break handle `*start_global_number+i` for the global break described by the string `global_breaks[i]`.

Note

The model must stop the simulation if the global break is enabled, see *Stopping RVISS* on page 4-18.

4.1.8 Register services

A model might have internal state that is to be exposed to the user. For example, a memory-mapped peripheral might want to provide a view of the state of the peripheral other than relying on the user examining the memory interface of the peripheral directly.

SimRdi_Manager has a unified register space, called *Uniregs*, that is split into blocks. These blocks are enumerated and named in `armulif/uniregs_registration_event.h`. The standard ARM coprocessors and cores occupy the first 32 blocks (0...31), and blocks 32 (`uniregs_armulator`) to 63 (`uniregs_armulator_top`) are reserved for use by simulators.

When a model wants to expose some registers it fills out one or more `SimRdi_Uniregs_Adverts` types. This type has the following members:

`x` The base advert which the model must fill in as defined above.

`description` A reasonably short description of the register set. This can be used by the autogenerator of register window tabs (see *Autogenerated register window tabs* on page 4-17).

`sdm_me` Reserved, and must be left `NULL`.

`block_num` The block to register against, and must be one of the block names defined in `uniregs_registration_event.h` between `uniregs_armulator` and `uniregs_armulator_top`.

`config_flags` The adverts in a block are sorted by `config_flags` and `UNIREGS_SORT_MASK`. If the `UNIREGS_DO_NOT_AUTO_GENERATE_REG_WIN` bit is set then the register is not automatically displayed (see *Autogenerated register window tabs* on page 4-17).

`len` The number of registers being exposed by this advert.

`desc` An array of descriptions for the registers being exposed by the model. The length of this array is specified by `len`.

`handle_for_function`

The handle that is passed to the `_SimReg` function.

`_SimReg` The function that reads and writes the registers. The type of this function is `SimabsRegisterAccess`:

```
typedef SIM_ERR SimabsRegisterAccess(void *handle,
                                     bool_int reg_read,
                                     uint32 reg_num,
                                     REGVAL *regval, uint16 size);
```

`handle` The `handle_for_function` from the advert.

`reg_read` `True` for a read, or `False` for a write.

`regval` The register value.

`size` The size of the register in bytes.

The register corresponding to description `desc[i] (0<=i<len)` is passed as `reg_num`, and is given by:

```
reg_num = (ad->reg_numbers == NULL)
            ? (i + ad->start_reg_number)
            : reg_number[i];
```

Therefore, if the register numbers in your model are not consecutive you can have SimRdi_Manager give you the correct numbers without having to map them.

The maximum size of registers is 32 bits and they can be read or written in host-endian format. Normally, using `memcpy` suffices, but a model can use `regval->reg32` for 32 bit quantities.

Note

A future version of SimRdi_Manager might enable you to change the maximum size of registers.

Returns `SIM_OK` on success, or `SIM_REGISTER_ACCESS` on error.

`read` **and** `write`

Must not be used by you, because they are used by the RVISS coprocessors. Leave these set to `NULL`.

Register definitions

The registers are described using the `Register_Definition` type whose members are:

`size` The size in bytes of the register.

`change_case` Set this to one of the following:

`EECHGCASE_UP`
: The uppercase version of the register symbol name is accepted in addition to the exact register name.

`EECHGCASE_DOWN`
: The lowercase version of the register symbol name is accepted in addition to the exact register name.

`EECHGCASE_NOT`
: Only the exact register name is accepted.

`type` One of `TYPE_SIGNED_CHAR`, `TYPE_UNSIGNED_CHAR`, `TYPE_SIGNED_SHORT_INT`, `TYPE_UNSIGNED_SHORT_INT`, `TYPE_SIGNED_LONG` or `TYPE_UNSIGNED_LONG`.

`name` The symbol name of the register. By convention it starts with the `@` symbol and must only contain alphanumeric characters, _, `@`, and `$`.

`buttonName` The name that appears in the register window if the model asks for it to be exposed. The `buttonName` must be padded to the right with spaces to make it a fixed length long. This means that the register values all line up when shown in the register window. Typically, a model pads the string to a length that is the same length as the longest unpadded `buttonName` that it wants to expose.

`flags` Set this to one of the following:

- **0** read/write
- **1** read-only
- **2** write-only

`reg_map` Reserved. Leave this set at zero.

Block numbers

Block numbers are used to group adverts for particular kinds of register together. Current block numbers are defined in `uniregs_registration_event.h`:

`uniregs_armulator`

Reserved.

`uniregs_general_regs_export`

Put in this block any register that does not seem to fit anywhere else.

`uniregs_model_debugging_registers`

This block is not intended for end users to see. Put registers in here that you find useful while debugging the model code but do not want end users seeing.

Note

For the registers in this model to be available you must set the `ARM_VERBOSE` environment variable as follows:

```
ARM_VERBOSE=simrdi_manager:show_all_blocks_in_regwin
```

If you have an RealView Connection Broker process running you must restart it to ensure that it recognizes the new value for `ARM_VERBOSE`.

`uniregs_cycle_counters`

This block contains the cycle counter for the model. For example, the cycle counter provided by the memory callback is always the top one in this block. This is because the cycle counter sorting bits of `config_flags` are all set and the counter gets sorted first. The cycle counters provided by the model are sorted last (corresponding to RealView Debugger `@stats_`*`symbolname`* symbols).

`uniregs_peripherals`

This block contains peripherals/models that are tied to a particular processor.

`uniregs_shared_peripherals`

This block contains peripherals that are shared between many cores on a multi-core platform.

`uniregs_shared_buses`

Registers relating to shared buses.

`uniregs_extensions`

Those modules that are not peripherals but are general extensions can put their registers here.

`uniregs_shared_extensions`

Modules that are extensions that are not tied to a particular core in multi-core systems must go here.

The symbols in blocks are only exported if they exist in the list `srpv->blocks_to_export`, that is an array of maximum length `srpv->blocks_to_export_len`. This is a list of block numbers terminated by a block number of -1.

During the debugging of models it might be useful to set the `ARM_VERBOSE` environment variable to `simrdi_manager:show_all_blocks_in_regwin`. This forces all blocks to be exported.

4.1.9 Register windows service (regwin)

The *register windows service* (regwin) service of SimRdiProcVec lets you define your own register tabs. Each advert can only register a single tab and it contains a pointer to a type RegWin that is a typedef to structure REG_WIN.

The REG_WIN structure has the following members:

tab_name
: The format *short_name*,*long_name*. *short_name* is what appears on the tab, *long_name* is what appears in the tool tip, or when you right-click on the tab.

lines
: An array of lines. The format for lines is:

  "_text"
  : An uninterpreted description line.

  "$+"
  : Expansion block (the little + button that you push to expose more registers), the default for this is normally closed. The following line must be a "_text" line that names this block.

  "$-"
  : Expansion block, normally open. Again the following line must be a "_text" line.

  "=name"
  : Corresponds to one of the register names that has been added through the register adverts, for example, @model_register. The name that appears in the window is the buttonName field of the Register_Definition structure.

    Note

    Each buttonName must be padded with spaces if you want the values to line up.

  "name"
  : Only the value appears.

  Note

  None of the strings in these arrays must contain tab or new-line characters.

line_cnt
: Number of lines in lines.

enum_cnt
: Reserved. Set to zero.

enum_list
: Reserved. Set to NULL.

Autogenerated register window tabs

SimRdi_Manager automatically generates an entry for the variables in a register tab that corresponds to a specific block when both these conditions exist:

- the `config_flags` of the variables do not have the following bit set (in `uniregs_registration_event.h`):

  `UNIREGS_DO_NOT_AUTO_GENERATE_REG_WIN`

- the block number appears in the `blocks_to_export` list in the `SimRdiProcVec`.

Each advert is converted to an expansion block, that is open by default, and the description field of `SimRdi_Uniregs_Advert` is used as the expansion name of the block.

Note

If you want to alter the order of tabs that appear in the window, then use the `reg_numbers` field of the advert to reorder them. The order in the `reg_numbers` array is the order they appear in the register window.

The adverts and, therefore, the order of expansion groups, can be changed by altering the bits of `config_flags & UNIREGS_SORT_MASK`.

Note

Currently only the `uniregs_cycle_counters` block has a special interpretation of the sort order of adverts, but in the future ARM Limited might add special interpretations to other blocks. Suggestions for the use of each block are provided in the comments in `uniregs_registration_event.h`.

If your model is to expose many registers, or requires expansion blocks, then your model must create its own register window. The autogenerate facilities are provided only as a way of preventing tab-proliferation.

4.1.10 Other members of SimRdiProcVec

SimRdiProcVec is the structure that SimRdi_Manager-aware modules see. This structure contains:

agent
: The agent handle for the system.

armulator_handle
: The handle for RVISS. The use of this is discouraged as it does not enable multiprocessor systems to work. Normally, models have other ways of obtaining the related core module handle and procvec (if they have access to a coredesc then it contains both).

armulator
: The procvec for RVISS.

sim_handle
: Some more advanced methods that callback through RealView Connection Broker require a handle. None of these methods are available for users in this release.

little_endian
: True if RealView Debugger believes that the target is little-endian.

really_little_endian
: True if the target is little-endian.

target_is_executable
: True if the target is currently executable. The user setting the PC or resetting the processor changes this from False to True. If your model changes the state of the model then it can change this to True or False as required.

Note

More elements are included, but these are either intended to be used solely by ARM Limited, or to be made available in future releases. Use of any of these features is not supported by ARM Limited, and there is no guarantee to support them in future releases. However, the calls and structures explicitly mentioned in this document are supported.

4.1.11 Stopping RVISS

To stop RVISS, your model must call the stop_simulation() member of SimRdiProcVec. Before your model calls this function, it must provide a reason why it is stopping. This is done by filling out the stop_info structure and by setting stop_reason_valid to True.

The STATUS_INFO structure is declared in tmgrem.h where more detailed information is available, and it has the following members:

detail
: Holds extra information depending on the value of the mode field.

mode
: Set this to one of the values specified by STATUS_MODE. Commonly used values are:

  SMODE_UNKNOWN
  : Unknown reason.

  SMODE_SIG
  : An exception, signal, or other event stopped RVISS. The detail field contains a reason for stopping. See *Built-in stop reasons* on page 4-20 for a list of built-in stop reasons.

  SMODE_SIG_MEM
  : An exception on memory stopped RVISS. The detail field contains a reason for stopping. See *Built-in stop reasons* on page 4-20 for a list of built-in stop reasons.

  SMODE_GLOBRK
  : A global break stopped RVISS. The detail field contains the global break handle as described in the section on global break adverts.

trip_page
: If you know the address of the reason for the stop then set this to 0. Otherwise set this to 0xFFFFFFFF.

trip_addr
: If you know the address of the reason for the stop then set this to the address. Otherwise set this to 0xFFFFFFFF.

If the target gets into a state where it cannot execute:

1. Set mode to SMODE_SIG.
2. Set detail to OSIG_BADSTATE (see Table 4-1 on page 4-20).
3. Clear the srpv->target_is_executable flag.

If your model outputs an error message through the host interface:

1. Set mode to SMODE_SIG.
2. Set detail to OSIG_ERROR_MSG (see Table 4-1 on page 4-20).

Built-in stop reasons

The built-in reasons for stopping begin with the prefix OSIG_, and they are declared in the enumeration GEN_SIGNALS in tmgrem.h. The reasons are listed in Table 4-1.

Table 4-1 Built-in stop reason values

Reason value	Description
OSIG_USER_HALT	Halted by user action
OSIG_EMU_STOP	Stop from emulator
OSIG_ILL_OP	Illegal instruction
OSIG_MEM_VIOL	Memory access violation
OSIG_TIME_OUT	Time-out from emulator
OSIG_NO_POWER	No target power detected
OSIG_BUSY	Target not responding because it is busy
OSIG_ERROR	Unknown error
OSIG_ERROR_MSG	Error from target
OSIG_RESET	Reset of target
OSIG_ABORT	Aborted
OSIG_BADSTATE	Bad state
OSIG_BUSERR	Bus error
OSIG_INT	Interrupt
OSIG_TRAP	Trap to use
OSIG_ANA_FULL	Analyzer full
OSIG_ANA_TRIG	Analyzer triggered
OSIG_GLOBRK	Global break detected
OSIG_BRANCH	Control flow breakpoint
OSIG_REGBRK	Register breakpoint

4.2 RVISS models

RVISS comprises a collection of models that simulate ARM architecture-based hardware. They enable you to benchmark, develop, and debug software before your hardware is available. See *Accuracy* on page 2-2 for information on the limitations of RVISS models.

4.2.1 Configuring models through ToolConf

RVISS models are configured through `ToolConf`. `ToolConf` is a database of tags and values that RVISS reads from configuration files (`.dsc` and `.ami` files) during initialization (see *ToolConf* on page 4-105).

A number of functions are provided for looking up values from this database. The full set of functions is defined in:

`install_directory``\RVARMulator\ARMulator\...\...\clx\toolconf.h`

For Windows replace \ with /.

All the functions take an opaque handle called a `toolconf`.

4.3 RVISS model insertion

Models must register themselves with RVISS, otherwise RVISS cannot call them when required.

4.3.1 Example 1: RVISS without the Mapfile and Tracer inserted

Figure 4-1 shows an example structure of RVISS. This example includes both Mapfile and Tracer, but neither of them are inserted.

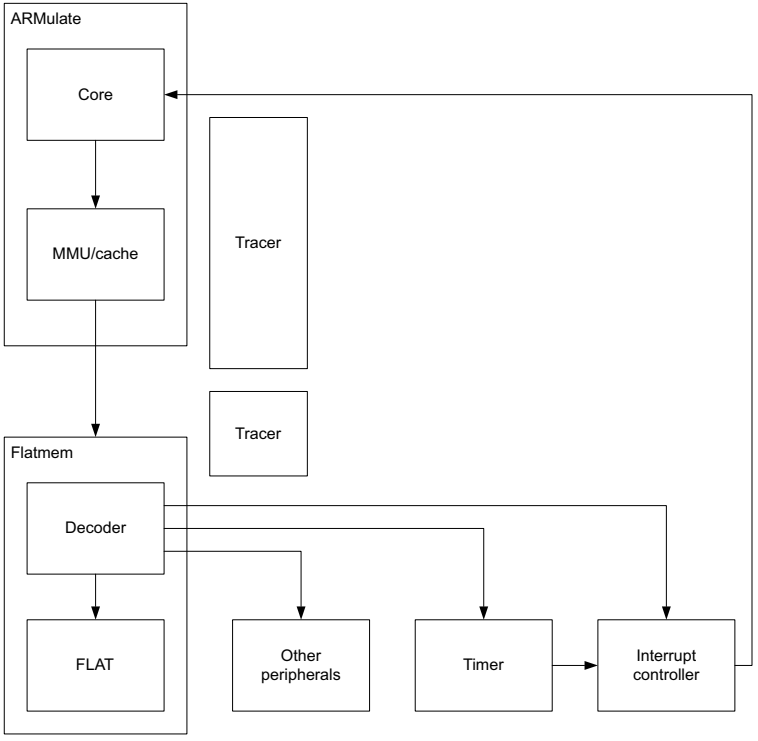

Figure 4-1 RVISS without the Mapfile and Tracer inserted

The links in this structure are produced as follows:

- During initialization, every peripheral, including Timer and Interrupt Controller, calls `bus->bus_registerPeripFunc(BusRegAct_Insert, regn)`. This creates the links from the decoder to the peripherals. See *ARMul_BusRegisterPeripFunc* on page 4-79 for details.

- During initialization, Timer calls `ARMulif_GetInterruptController`, and Interrupt Controller calls `ARMulif_InstallNewInterruptController` to create the link from Timer to Interrupt Controller. You can find the prototypes for these functions in `armul_askrdi.h`.
- The remaining links are created by RVISS itself.

At run time, Interrupt Controller calls `ARMulif_SetSignal` (see *ARMulif_SetSignal* on page 4-61) to use the link from Interrupt Controller to the core.

4.3.2 Example 2: RVISS with Mapfile inserted, and Tracer inserted in one link

Figure 4-2 shows the structure with both Mapfile and Tracer inserted.

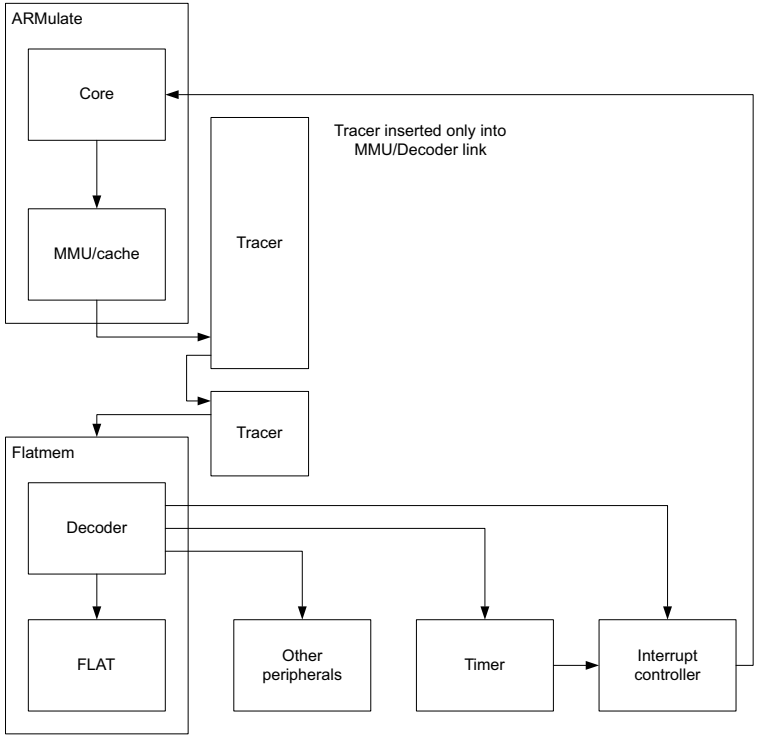

Figure 4-2 RVISS with Mapfile inserted, and Tracer inserted in one link

The links in this figure are created in the same way as in Example 1, except that:

- During initialization, Mapfile calls `ARMulif_QueryMemInterface` and `ARMul_InsertMemInterface` to insert itself in the link between MMU/Cache and Flatmem.

- Tracer calls `ARMulif_QueryMemInterface` and `ARMul_InsertMemInterface` to insert itself in the link between MMU/Cache and Flatmem. Tracer can do this at any time.

  Tracer can also call `ARMul_RemoveMemInterface` to remove itself at any time.

You can find the prototypes for these functions in `armul_askrdi.h` and `armul_mem.h`.

4.3.3 Example 3: RVISS with Mapfile inserted, and Tracer inserted in two links

Figure 4-3 shows the structure with the Mapfile inserted, and the Tracer inserted in two links.

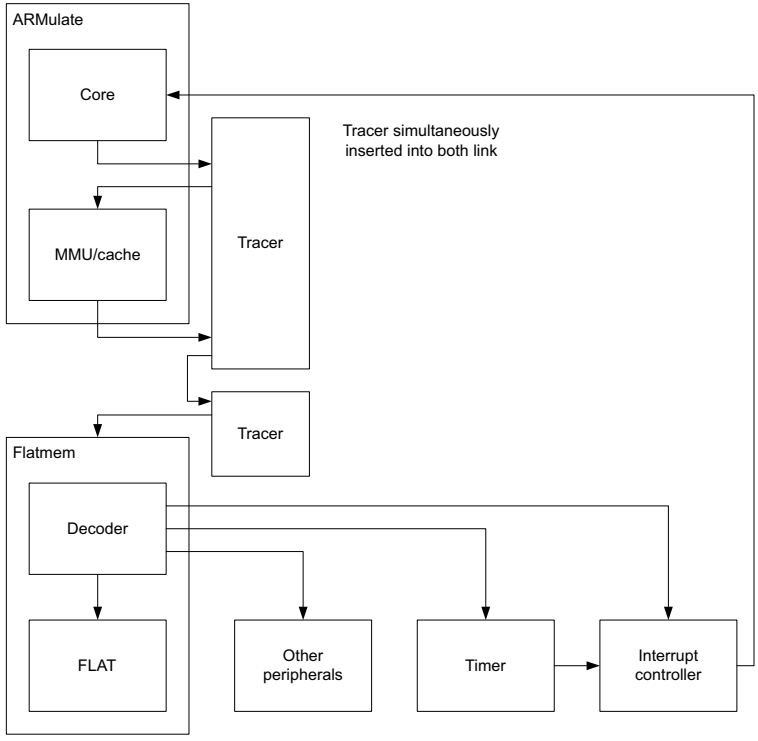

Figure 4-3 RVISS with Mapfile inserted, and Tracer inserted in two links

The links in Figure 4-3 on page 4-24 are created in the same way as shown in *Example 2: RVISS with Mapfile inserted, and Tracer inserted in one link* on page 4-23, except that:

- Tracer calls `ARMulif_QueryMemInterface` and `ARMul_InsertMemInterface` a second time to insert itself in the link between Core and MMU/Cache. Tracer can do this at any time.

  Tracer can also call `ARMul_RemoveMemInterface` to remove itself at any time.

4.4 Communicating with the core

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

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

- *ARMulif_GetReg* on page 4-27
- *ARMulif_SetReg* on page 4-28
- *ARMulif_GetPC and ARMulif_GetR15* on page 4-28
- *ARMulif_SetPC and ARMulif_SetR15* on page 4-29
- *ARMulif_GetCPSR* on page 4-29
- *ARMulif_SetCPSR* on page 4-29
- *ARMulif_GetSPSR* on page 4-30
- *ARMulif_SetSPSR* on page 4-30.

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

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

- *ARMulif_ThumbBit* on page 4-31
- *ARMulif_GetMode* on page 4-31.

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

- *ARMulif_CPRead* on page 4-32
- *ARMulif_CPWrite* on page 4-33.

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

- *ARMulif_SetConfig* on page 4-34.

Note

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

4.4.1 Mode numbers

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 here:

- USER32MODE
- FIQ32MODE
- IRQ32MODE
- SVC32MODE
- ABORT32MODE
- UNDEF32MODE
- SYSTEM32MODE

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

4.4.2 ARMulif_GetReg

This function reads a register for a specified processor mode.

Syntax

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

where:

mdesc is the handle for the core.

mode is the processor mode. Values for mode are defined in armdefs.h (see *Mode numbers*).

reg is the register to read. Valid values are 0 to 14 for registers r0 to r14, PC, or CPSR.

Return

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

4.4.3 ARMulif_SetReg

This function writes a register for a specified processor mode.

Syntax

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

where:

mdesc is the handle for the core.

mode is the processor mode. Mode numbers are defined in `armdefs.h` (see *Mode numbers* on page 4-27).

reg is the register to write. Valid values are 0 to 14 for registers r0 to r14, PC, or CPSR.

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

Usage

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

4.4.4 ARMulif_GetPC and ARMulif_GetR15

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

Syntax

```
ARMword ARMulif_GetPC(RDI_ModuleDesc *mdesc)

ARMword ARMulif_GetR15(RDI_ModuleDesc *mdesc)
```

where:

mdesc is the handle for the core.

Return

This function returns the value of the pc.

4.4.5 ARMulif_SetPC and ARMulif_SetR15

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

Syntax

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

where:

mdesc is the handle for the core.

value is the value to be written to the pc.

4.4.6 ARMulif_GetCPSR

This function reads the CPSR.

Syntax

```
ARMword ARMulif_GetCPSR(RDI_ModuleDesc *mdesc)
```

where:

mdesc is the handle for the core.

Return

The function returns the value of the CPSR.

4.4.7 ARMulif_SetCPSR

This function writes a value to the CPSR.

Syntax

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

where:

mdesc is the handle for the core.

value is the value to be written to the CPSR.

4.4.8 ARMulif_GetSPSR

This function returns the current contents of the SPSR for a specified processor mode.

Syntax

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

where:

mdesc is the handle for the core.

mode is the processor mode for the SPSR you want to read.

User mode

`ARMulif_GetSPSR` returns the current contents of the CPSR if *mode* is `USER32MODE`.

4.4.9 ARMulif_SetSPSR

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

Syntax

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

where:

mdesc is the handle for the core.

mode is the processor mode for the SPSR you want to write.

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

User mode

`ARMulif_SetSPSR` does nothing if *mode* is `USER32MODE`.

4.4.10 ARMulif_ThumbBit

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

Syntax

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

where:

mdesc is the handle for the core.

4.4.11 ARMulif_GetMode

This function reads the current processor mode.

Syntax

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

where:

mdesc is the handle for the core.

4.4.12 ARMulif_CPRead

This function calls the `read` method for a coprocessor.

Syntax

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

where:

mdesc is the handle for the core.

cpnum is the number of the coprocessor.

reg is the number of the coprocessor register to read from, as indexed by `CRn` in an `LDC` or `STC` instruction.

data is a pointer for the data read from the coprocessor register. The number of words transferred, and the order of the words, is coprocessor dependent.

Return

The function must return:

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

4.4.13 ARMulif_CPWrite

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

Syntax

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

where:

mdesc is the handle for the core.

cpnum is the number of the coprocessor.

reg is the number of the coprocessor register to read from, as indexed by CR*n* in an LDC or STC instruction.

data is a pointer for the data read from the coprocessor register. The number of words transferred, and the order of the words, is coprocessor dependent.

Return

The function must return:

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

4.4.14 ARMulif_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.

Syntax

```
void ARMulif_SetConfig(RDI_ModuleDesc *mdesc,
                       ARMword bitsToChange, ARMword newValue)
```

where:

mdesc is the handle for the core.

bitsToChange is a bitmask of the config bits to change.

newValue 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 *bitsToChange* it must not be set in *newValue*. For example, to set bit[1] and clear bit[0]:

bitsToChange 0x03 (0b00000011)

newValue 0x02 (0b00000010)

Example

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

You can use the following call to obtain the current settings of the configuration pins, without modifying them:

```
currentConfig = ARMulif_SetConfig(state, 0, 0)
```

4.5 Basic model interface

This section describes the basic interface model:

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

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

- *Declaration of a private state data structure*
- *Model initialization* on page 4-36
- *Model finalization* on page 4-36.

See also *Initialization, finalization, and state macros* on page 3-7.

4.5.1 Declaration of a private state data structure

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

To declare a state data structure, use the `BEGIN_STATE_DECL` and `END_STATE_DECL` macros as follows:

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

This declares a structure:

```
typedef struct YourModelState
```

This structure contains:

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

4.5.2 Model initialization

The BEGIN_INIT() and END_INIT() macros form the start and finish of the initialization function for the model. The initialization function is called:

- during RVISS initialization
- whenever RealView Debugger is downloading a new image.

The following local variables are provided in the initialization function:

- `bool coldboot`

  TRUE if RVISS is initializing, FALSE if RealView Debugger is downloading a new image.

- `YourModelState *state`

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

In the initialization function, your model must:

- initialize any private data
- install any callbacks.

4.5.3 Model finalization

The BEGIN_EXIT() and END_EXIT() macros form the start and finish of the finalization function for the model. The finalization function is called when RVISS is closing down.

The following local variable is provided in the finalization function:

```
YourModelState *state
```

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

The END_EXIT() macro frees memory allocated for state.

4.6 The memory interface

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

Because there are many core processor types, there are many memory type variants. The memory initialization function is told which type it must provide (see *Memory model initialization function* on page 4-41). A model must refuse to initialize in the case of an unrecognized memory type variant.

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 must use `TransChangeUpcall()` to track **nTRANS**. You can find the prototype for `TransChangeUpcall` in `armul_mem.h`.

4.6.1 Memory type variants

The memory type variants are defined in the `ARMul_MemInterface` structure in `armul_mem.h`.

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. The halfword instruction fetches can be sequential.

This can indicate to a memory model that most accesses are halfword-instruction-sequential rather than the usual word-instruction-sequential.

Note

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

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

The file `flatmem.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`.
- cycle, where `acc_MREQ(acc)` is `FALSE`.

A non-sequential 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 *Macros for access types* on page 4-49) 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[7:0] of the data, bit[1] to bits[15:8], bit[2] to bits[23:16], and bit[3] to bits[31:24].

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
```

4.7 Memory model interface

The memory model interface is defined in the file `armul_mem.h`, which is included from `armul_defs.h`. All memory access 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.

At initialization time, the initialization function registers a number of functions in the memory interface structure, `ARMul_MemInterface` in `armul_mem.h`.

For details of the initialization function, see:

- *Memory model initialization function* on page 4-41.

For details of the basic function entries, see:

- *armul_ReadClock* on page 4-42
- *armul_GetCycleLength* on page 4-42
- *armul_ReadCycles* on page 4-43
- *armul_MemAccess* on page 4-44.

For details of the functions required for some processors, but not for others, see:

- *armul_MemAccess2* on page 4-45
- *armul_MemAccAsync* on page 4-46
- *armul_HarvardMemAccess* on page 4-47.

Type definitions for these functions are in `armul_mem.h`.

Note

`armul_mem.h` contains several type definitions for several functions that are *not* used by RVISS. You do not have to supply these functions.

4.7.1 Memory model initialization function

A memory model must export 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

```
void  ARMul_Error armul_MemInit(struct ARMul_State *state,
                                ARMul_MemInterface *interf,
                                /* ARMul_MemType variant, */
                                toolconf your_config, toolconf core_config)
```

where:

state is a pointer to the RVISS state.

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

variant is the interface variant. See the `ARMul_MemType` enumeration in `armul_mem.h`. See *Memory type variants* on page 4-37 for a description of the variants.

your_config is the configuration database for your model or models.

core_config is the configuration database for the core.

Return

This function returns either:

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

The error must be passed through `Hostif_RaiseError()` for formatting (see *Hostif_RaiseError* on page 4-87).

Usage

The initialization must 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. RVISS passes this handle to all the access functions it calls.

4.7.2 armul_ReadClock

This function must return the elapsed time in microseconds since the simulation model reset.

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

Syntax

```
ARMTime armul_ReadClock(void *handle)
```

where:

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

Return

This function returns an `ARMTime` value representing the elapsed time in microseconds. The default type of `ARMTime` is **`unsigned long`**. `ARMTime` is defined in `armul_types.h`.

Usage

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

4.7.3 armul_GetCycleLength

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

You must implement this function, even if the implementation is very simple. You define the function name yourself.

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.

4.7.4 armul_ReadCycles

The `read_cycles` entry in the `ARMul_MemInterface` structure is a pointer to an `armul_ReadCycles()` function. This function must calculate the total cycle count since the simulation model reset.

You must implement this function, even if the implementation is very simple. You define the function name yourself.

Syntax

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

where:

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

Return

RVISS calls this function each time RealView Debugger reads the counters. It must calculate the total cycle count and returns a pointer to the `ARMul_Cycles` structure that contains the cycle counts. The `ARMul_Cycles` structure is defined in `armul_mem.h`.

Usage

A model can keep count of the accesses made to it by RVISS by providing this function. The value of the `CoreCycles` field in `ARMul_Cycles` is provided by RVISS, not by 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 4-75 for a description of `ARMul_Time()`.

4.7.5 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. You define the function name yourself.

Syntax

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

where:

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

address is the value on the address bus.

data is a pointer to the data for the memory access. See *Data for reads and writes* on page 4-48 for details.

access_type encodes the type of cycle. On some processors, for example, cached processors, some of the signals are not valid. See *Macros for access types* on page 4-49 for details of the macros for determining access type.

Return

The function returns:

1 indicates successful completion of the cycle

0 tells the processor to busy-wait and try the access again next cycle

–1 signals an abort

–2 indicates that an address was not decoded by a peripheral model (see *Reference peripherals* on page 4-111).

Note

Memory models must not return –2. Only a peripheral that has registered an address range with a bus-decoder can return –2.

4.7.6 armul_MemAccess2

This function is required for ARM8 models.

Syntax

```
int armul_MemAccess2(void *handle, ARMword address, ARMword *data,
                     ARMul_acc access_type)
```

where:

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

address is the value on the address bus.

data is a pointer to the data for the memory access. See *Data for reads and writes* on page 4-48 for details.

access_type encodes the type of cycle. On some processors, for example, cached processors, some of the signals are not valid. See *Macros for access types* on page 4-49 for details of the macros for determining access type.

Return

The function returns:

1 Indicates successful completion of the cycle.

0 Tells the processor to busy-wait and try the access again next cycle.

–1 Signals an abort.

2 Signals successful return of a single word of a doubleword load. To load the second word, the caller increments *address* by 4 and issues a single word load.

4.7.7 armul_MemAccAsync

This is a memory access function used by ARM10, ARM11 and XScale models.

Syntax

```
int armul_MemAccAsync(void *handle, ARMword address, ARMword *data,
ARMul_acc acc, ARMTime *abs_time)
```

where:

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

address is the value on the address bus.

data is a pointer to the data for the memory access. See *Data for reads and writes* on page 4-48 for details.

acc encodes the type of cycle. On some processors, for example, cached processors, some of the signals are not valid. See *Macros for access types* on page 4-49 for details of the macros for determining access type.

abs_time is the absolute time since reset.

Return

The function returns:

1 indicates successful completion of the cycle

0 tells the processor to busy-wait and try the access again next cycle

–1 signals an abort.

4.7.8 armul_HarvardMemAccess

This is the memory access function used for true Harvard models, where both busses present the required access parameters in the same function call.

Syntax

```
void armul_HarvardMemAccess(void *handle, ARMword address1, ARMword *data1,
                            ARMul_acc access1, int *return1, ARMword address2,
                            ARMword *data2, ARMul_acc access2, int *return2)
```

where:

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

address1 is the value on the data address bus.

data1 is a pointer to the data for the data memory access. See *Data for reads and writes* on page 4-48 for details.

access1 encodes the type of cycle for the data memory access. On some processors, for example, cached processors, some of the signals are not valid. See *Macros for access types* on page 4-49 for details of the macros for determining access type.

return1 is the return value for the data memory access:

- **1** indicates successful completion of the cycle
- **0** tells the processor to busy-wait and try the access again next cycle
- **–1** signals an abort.

address2 is the value on the instruction address bus.

data2 is a pointer to the data for the instruction memory access. See *Data for reads and writes* on page 4-48 for details.

access2 encodes the type of cycle for the instruction memory access. On some processors, for example, cached processors, some of the signals are not valid. See *Macros for access types* on page 4-49 for details of the macros for determining access type.

return2 is the return value for the instruction memory access:

- **1** indicates successful completion of the cycle
- **0** tells the processor to busy-wait and try the access again next cycle
- **–1** signals an abort.

4.7.9 Data for reads and writes

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

Note

Your model must ensure that the value written is the correct width.

The model can ignore the alignment of the address passed to it because this is handled by RVISS. However, it must present the bytes of the word in the correct order for the byte order of the processor. Your model can determine this by using either a `ConfigChangeUpcall()` upcall or `ARMulif_SetConfig()` (see *Communicating with the core* on page 4-26).

`armul_defs.h` provides a flag variable macro named `HostEndian`. `HostEndian` is `TRUE` if RVISS is running on a big-endian machine. See the `flatmem.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 might have to be shortened for a byte or halfword store.

As with reads, byte order must be handled correctly.

4.7.10 Macros for access types

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, these determine whether the current access is a read or a 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`, `BITS_32`, or `BITS_64` depending on whether a byte, halfword, word or doubleword is being accessed.

4.8 Coprocessor model interface

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

- *ARMulif_InstallCoprocessorV5* on page 4-51
- *LDC* on page 4-52
- *STC* on page 4-53
- *MRC* on page 4-54
- *MCR* on page 4-55
- *MRC* on page 4-54
- *MCR* on page 4-55
- *MCRR* on page 4-56
- *MRRC* on page 4-57
- *CDP* on page 4-58.

Caution

Some coprocessors have registers that are write-only. The value written to these registers must be a specific value. If an incorrect value is written to these registers in a model, the result is unpredictable, and might not follow what happens in the hardware.

In addition, two functions are provided that enable RealView Debugger to read and write coprocessor registers. They are:

- *read* on page 4-59
- *write* on page 4-60.

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

4.8.1 ARMulif_InstallCoprocessorV5

Use this function to register a coprocessor handler.

This function is prototyped in `armul_copro.h`.

Syntax

```
unsigned ARMulif_InstallCoprocessorV5(RDI_ModuleDesc *mdesc, unsigned number,
                                      struct ARMul_CoprocessorV5 *cpv5, void *handle)
```

where:

mdesc is the handle for the core.

number is the coprocessor number.

cpv5 is a pointer to the coprocessor interface structure.

handle is a pointer to private data to pass to each coprocessor function.

Return

This function returns either:

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

The error must be passed through `Hostif_RaiseError()` for formatting (see *Hostif_RaiseError* on page 4-87).

4.8.2 LDC

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

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

- ARMul_CP_FIRST — indicates that this is the first time the coprocessor model has been called for this instruction.
- ARMul_CP_BUSY — indicates that this is a subsequent call, after the first call was busy-waited.
- ARMul_CP_INTERRUPT — warns the coprocessor that the ARM processor is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction is retried later, in which case the *type* is reset to ARMul_CP_FIRST.
- ARMul_CP_TRANSFER — indicates that the ARM processor is about to perform the load.
- ARMul_CP_DATA — indicates that valid data is included in *data*.

instr the current opcode.

data is a pointer to the data being loaded to the coprocessor from memory.

Return

The function must return one of:

- ARMul_CP_INC, to request more data from the core (only in response to ARMul_CP_FIRST, ARMul_CP_BUSY, or ARMul_CP_DATA).
- ARMul_CP_DONE, to indicate that the coprocessor operation is complete (only in response to ARMul_CP_DATA).
- ARMul_CP_BUSY, to indicate that the coprocessor is busy (only in response to ARMul_CP_FIRST or ARMul_CP_BUSY).
- ARMul_CP_CANT, to indicate that the instruction is not supported, or the specified register cannot be accessed (only in response to ARMul_CP_FIRST or ARMul_CP_BUSY).

- ARMUL_CP_LAST, to indicate that the next load is the last in the sequence. This is only required for ARM9.

4.8.3 STC

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

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

ARMul_CP_FIRST	indicates that this is the first time the coprocessor model has been called for this instruction.
ARMul_CP_BUSY	indicates that this is a subsequent call, after the first call was busy-waited.
ARMul_CP_INTERRUPT	warns the coprocessor that the ARM processor is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction is retried later. In that case the *type* is reset to ARMul_CP_FIRST.
ARMul_CP_DATA	indicates that the coprocessor must return valid data in **data*.

instr is the current opcode.

data is a pointer to the location of the data being saved to memory.

Return

The function must return one of:

- ARMul_CP_INC, to indicate that there is more data to transfer to the core (only in response to ARMul_CP_FIRST, ARMul_CP_BUSY, or ARMul_CP_DATA).
- ARMul_CP_DONE, to indicate that the coprocessor operation is complete (only in response to ARMul_CP_DATA).
- ARMul_CP_BUSY, to indicate that the coprocessor is busy (only in response to ARMul_CP_FIRST or ARMul_CP_BUSY).

- ARMul_CP_CANT, to indicate that the instruction is not supported, or the specified register cannot be accessed (only in response to ARMul_CP_FIRST or ARMul_CP_BUSY).
- ARMUL_CP_LAST, to indicate that the next save is the last in the sequence. This is only required for ARM9.

4.8.4 MRC

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

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

ARMul_CP_FIRST	indicates that this is the first time the coprocessor model has been called for this instruction.
ARMul_CP_BUSY	indicates that this is a subsequent call, after the first call was busy-waited.
ARMul_CP_INTERRUPT	warns the coprocessor that the ARM processor is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction is retried later, in which case the *type* is reset to ARMul_CP_FIRST.
ARMul_CP_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_CP_DONE, to indicate that the coprocessor operation is complete, and valid data has been returned to **data*
- ARMul_CP_BUSY, to indicate that the coprocessor is busy

- ARMul_CP_CANT, to indicate that the instruction is not supported, or the specified register cannot be accessed.

4.8.5 MCR

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

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

- ARMul_CP_FIRST — indicates that this is the first time the coprocessor model has been called for this instruction.
- ARMul_CP_BUSY — indicates that this is a subsequent call, after the first call was busy-waited.
- ARMul_CP_INTERRUPT — warns the coprocessor that the ARM processor is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction is retried later, in which case the *type* is reset to ARMul_CP_FIRST.
- ARMul_CP_DATA — indicates valid data is included in *data*.

instr is the current opcode.

data is a pointer to the data being transferred to the coprocessor.

Return

The function must return one of:

- ARMul_CP_DONE, to indicate that the coprocessor operation is complete
- ARMul_CP_BUSY, to indicate that the coprocessor is busy
- ARMul_CP_CANT, to indicate that the instruction is not supported, or the specified register cannot be accessed.

4.8.6 MCRR

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

The function must return ARMul_CP_CANT if:

- the requested coprocessor register does not exist
- the requested coprocessor register cannot be written to
- the coprocessor is ARMv4T or earlier.

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

ARMul_CP_FIRST	indicates that this is the first time the coprocessor model has been called for this instruction.
ARMul_CP_BUSY	indicates that this is a subsequent call, after the first call was busy-waited.
ARMul_CP_INTERRUPT	warns the coprocessor that the ARM processor is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction is retried later, in which case the *type* is reset to ARMul_CP_FIRST.
ARMul_CP_DATA	indicates valid data is included in *data*.

instr is the current opcode.

data is a pointer to the data being transferred to the coprocessor.

Return

The function must return one of:

- ARMul_CP_DONE, to indicate that the coprocessor operation is complete
- ARMul_CP_BUSY, to indicate that the coprocessor is busy
- ARMul_CP_CANT, to indicate that the instruction is not supported, or the specified register cannot be accessed.

4.8.7 MRRC

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

The function must return ARMul_CP_CANT if:

- the requested coprocessor register does not exist
- the requested coprocessor register cannot be read from
- the coprocessor is ARMv4T or earlier.

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

ARMul_CP_FIRST	indicates that this is the first time the coprocessor model has been called for this instruction.
ARMul_CP_BUSY	indicates that this is a subsequent call, after the first call was busy-waited.
ARMul_CP_INTERRUPT	warns the coprocessor that the ARM processor is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction is retried later, in which case the *type* is reset to ARMul_CP_FIRST.
ARMul_CP_DATA	indicates valid data is included in *data*.

instr is the current opcode.

data is a pointer to the data being transferred from the coprocessor.

Return

The function must return one of:

- ARMul_CP_DONE, to indicate that the coprocessor operation is complete
- ARMul_CP_BUSY, to indicate that the coprocessor is busy
- ARMul_CP_CANT, to indicate that the instruction is not supported, or the specified register cannot be accessed.

4.8.8 CDP

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

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

- ARMul_CP_FIRST — indicates that this is the first time the coprocessor model has been called for this instruction.
- ARMul_CP_BUSY — indicates that this is a subsequent call, after the first call was busy-waited.
- ARMul_CP_INTERRUPT — warns the coprocessor that the ARM processor is about to service an interrupt, so the coprocessor must discard the current instruction. Usually, the instruction is retried later, in which case the *type* is reset to ARMul_CP_FIRST.

instr is the current opcode.

data is not used.

Return

The function must return one of:

- ARMul_CP_DONE, to indicate that the coprocessor operation is complete
- ARMul_CP_BUSY, to indicate that the coprocessor is busy
- ARMul_CP_CANT, to indicate that the instruction is not supported.

4.8.9 read

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

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

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

instr is not used.

value is a pointer to the location of the data to be read from the coprocessor.

Return

The function must return one of:

- ARMul_CP_DONE, to indicate that the coprocessor operation is complete
- ARMul_CP_CANT, to indicate that the register is not supported.

Usage

This function is called by RealView Debugger.

4.8.10 write

This function enables RealView Debugger to write to a coprocessor register.

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_CP_CANT.

Syntax

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

where:

handle is the handle from ARMulif_InstallCoprocessorV5.

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

instr is not used.

value is a pointer to the location of the data that is to be written to the coprocessor.

Return

The function must return one of:

- ARMul_CP_DONE, to indicate that the coprocessor operation is complete
- ARMul_CP_CANT, to indicate that the register is not supported.

Usage

This function is called by RealView Debugger.

4.9 Exceptions

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

- *ARMulif_SetSignal*
- *ARMulif_GetProperty* on page 4-62.

4.9.1 ARMulif_SetSignal

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

Syntax

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

where:

mdesc is the handle for the core.

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

`RDIPropID_ARMSignal_IRQ`
: Assert an interrupt.

`RDIPropID_ARMSignal_FIQ`
: Assert a fast interrupt.

`RDIPropID_ARMSignal_RESET`
: Assert the reset signal. The core resets, and does not restart until the reset signal is de-asserted.

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

`RDIPropID_ARMSignal_HighException`
: Set the base location of exception vectors.

`RDIPropID_ARMSignal_BranchPredictEnable`
: (ARM10 only)

`RDIPropID_ARMSignal_LDRSetTBITDisable`
: (ARM10 only)

`RDIPropID_ARMSignal_WaitForInterrupt`
: (ARM10 and XScale only)

`RDIPropID_ARMSignal_DebugState`
: Enter or exit debug state.

RDIPropID_ARMulProp_CycleDelta
: Wait the core for a specified number of cycles.

RDIPropID_ARMulProp_Accuracy
: Select the modeling accuracy, as a percentage in the range 0% to 100%. Currently this only affects ARM10 models. A setting less than 50% turns of interlock modeling. RVISS runs faster with interlock modeling turned off, but cycling count accuracy is reduced.

sigstate For signals, you must give *sigstate* one of the following values:

FALSE Signal off

TRUE Signal on.

For properties, you must give *sigstate* an integer value.

Note

For information about signalling interrupts when using an interrupt controller see *Interrupt controller* on page 4-111.

4.9.2 ARMulif_GetProperty

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

Syntax

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

where:

mdesc is the handle for the core.

id is the signal or property to read. *id* can be any one of:

RDIPropID_ARMSignal_IRQ
: TRUE if the interrupt signal is asserted.

RDIPropID_ARMSignal_FIQ
: TRUE if the fast interrupt signal is asserted.

RDIPropID_ARMSignal_RESET
: TRUE if the reset signal is asserted.

RDIPropID_ARMSignal_BigEnd
: TRUE if the bigend signal is asserted.

`RDIPropID_ARMSignal_HighException`

`TRUE` if the vector table is at `0xFFFF0000`.

`RDIPropID_ARMSignal_BranchPredictEnable`

(ARM10 only)

`RDIPropID_ARMSignal_LDRSetTBITDisable`

(ARM10 only)

`RDIPropID_ARMSignal_WaitForInterrupt`

(ARM10 and XScale only)

`RDIPropID_ARMulProp_CycleCount`

Count of the number of cycles executed since initialization.

`RDIPropID_ARMulProp_RDILog`

Current setting of the logging level. Generally, this is zero if logging is disabled, and nonzero if it is enabled.

`RDIPropID_ARMSignal_ProcessorProperties`

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

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

4.10 Events

RVISS 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 details depends on the event.

The core RVISS generates some example events, defined in `armdefs.h`. They are divided into the following groups:

- events from the ARM processor core, listed in Table 4-3 on page 4-65
- events from the MMU and cache (not on StrongARM-110), listed in Table 4-2
- events from the prefetch unit (ARM8-based processors only), listed in Table 4-4 on page 4-65
- configuration change events, listed in Table 4-6 on page 4-66.

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 are handled in the same way. User defined events must have values between `UserEvent_Base` (`0x100000`) and `UserEvent_Top` (`0x1FFFFF`).

You can catch events by installing an event handler (see *Event handler* on page 4-72). You can raise an event by calling `ARMulif_RaiseEvent()` (see *ARMulif_RaiseEvent* on page 4-67).

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

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

Table 4-3 Events from the ARM processor core

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

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

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

Table 4-5 Debug events

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

Table 4-6 Config events

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

4.10.1 ARMulif_RaiseEvent

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

Syntax

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

where:

mdesc is the handle for the core.

event is one of the event numbers defined in Table 4-2 on page 4-64, Table 4-3 on page 4-65, Table 4-4 on page 4-65, or Table 4-5 on page 4-66.

data1 is the first word of the event.

data2 is the second word of the event.

4.11 Handlers

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

You must provide implementations of the event handlers if you want to use them in your own models. For examples, see the implementations in the models supplied by ARM Limited.

You can use event handlers to avoid having to check state values on every access. For example, a peripheral model is expected to present the ARM core with data in the correct byte order for the value of the ARM processor **bigend** signal. A peripheral model can attach to the `EventHandler()` (see *Event handler* on page 4-72) to be informed when this signal changes.

4.11.1 Exception handler

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

Syntax

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

where:

handle is the handle passed to `ARMulif_InstallExceptionHandler`.

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

`((ARMul_Event *)data)->event`
: is the core event causing the exception (see Table 4-3 on page 4-65).

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

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

Usage

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

Note

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

Install the exception handler using:

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

The new exception handler is installed in one of the following ways:

- On models in the ARM7, ARM9 and StrongARM families, the new exception handler is added to the end of the list. The list is iterated over and all exception handlers are called. If any return `true`, the built-in exception handlers are not run.

- On models in the ARM10, ARM11 and XScale families, the new exception handler is added to the start of the list. When an exception occurs, the core iterates over the list from the beginning. This means that the handler that was installed last will be called first. The core stops iterating if a handler returns `true`.

Remove the exception handler using:

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

4.11.2 Unknown information handler

The unknown information function is called if RVISS cannot handle an `RDI_InfoProc` request itself. It returns an `RDIError` value. This function can be used by a model extending the interface between RVISS and RealView Debugger.

Syntax

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

where:

handle is the handle passed to `ARMulif_InstallUnkRDIInfoHandler`.

type is the `RDI_InfoProc` subcode. These are defined in `rdi_info.h`. See *Usage* for some examples.

arg1/arg2 are arguments passed to the handler from RVISS.

Usage

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

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

`RDIInfo_Target`

This enables models to declare how to extend the functionality of the target.

`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 logging level value.

`RDIRequestCyclesDesc`

This enables models to extend the list of counters provided by RealView Debugger `@stats_symbolname` symbols. Models call `ARMul_AddCounterDesc()` (see *General purpose functions* on page 4-76) to declare each counter in turn. It is essential that the model also trap the `RDICycles` 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 4-76) for each counter in turn, in the same order as they were declared.

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

Install the handler using:

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

Remove the handler using:

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

Example

The `semihost.c` model supplied with RVISS uses the `UnkRDIInfoUpcall()` to interact with RealView Debugger:

`RDIErrorP` returns errors raised by the program running under RVISS to RealView Debugger.

`RDISet_Cmdline` finds the command line set for the program by RealView Debugger.

`RDIVector_Catch` intercepts the hardware vectors.

4.11.3 Event handler

This handler catches RVISS events (see *Events* on page 4-64).

Syntax

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

where:

handle is the handle passed to `ARMulif_InstallEventHandler`.

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

`((ARMul_Event *)data)->event`
: is one of the event numbers defined in Table 4-2 on page 4-64, Table 4-3 on page 4-65, and Table 4-4 on page 4-65.

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

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

Usage

Install the handler using:

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

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

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

Remove the handler using:

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

Example handler installation

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

4.12 Memory access functions

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

Note

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

4.12.1 Reading from a given address

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

Syntax

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

ARMword ARMulif_ReadHalfword(RDIModuleDesc *mdesc, ARMword address)

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

where:

mdesc is the handle for the core.

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

Return

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

4.12.2 Writing to a specified address

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

Syntax

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

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

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

where:

mdesc is the handle for the core.

address is the address in simulated memory to write to.

data is the word or byte to write.

4.13 Event scheduling functions

The following functions enable you to schedule or remove events:

- *ARMulif_ScheduleTimedFunction*
- *ARMulif_DescheduleTimedFunction*.

4.13.1 ARMulif_ScheduleTimedFunction

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

Syntax

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

where:

mdesc is the handle for the core.

tcb is a handle for you to use if you want to remove the function from the scheduled memory cycle based event.

Note

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

4.13.2 ARMulif_DescheduleTimedFunction

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

Syntax

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

where:

mdesc is the handle for the core.

tcb is the handle supplied by `ARMulif_ScheduleTimedFunction` when the event was first set up.

4.14 General purpose functions

This section describes the general purpose RVISS functions. They include functions to access processor properties, add counter descriptions and values, stop RVISS and execute code.

4.14.1 ARMul_AddCounterDesc

The `ARMul_AddCounterDesc()` function adds new counters to the RealView Debugger `@stats_symbolname` symbols.

Syntax

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

where:

handle is no longer used.

arg1/*arg2* are the arguments passed to the `UnkRDIInfoUpcall()`.

name is a string that names the statistic counter. The string must be less than 32 characters long.

Return

The function returns one of:

- `RDIError_BufferFull`
- `RDIError_UnimplementedMessage`.

Usage

When RVISS receives an `RDIRequestCycleDesc()` call from RealView Debugger, it uses the `UnkRDIInfoUpcall()` (see *Unknown information handler* on page 4-70) 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 RealView Debugger calculates the statistics increments.

4.14.2 ARMul_AddCounterValue

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

Syntax

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

where:

handle is no longer used.

arg1/*arg2* are the arguments passed to the `UnkRDIInfoUpcall()`.

is64 denotes whether the counter is a pair of 32-bit words making a 64-bit counter (least significant word first), or a single 32-bit value. This enables modules to provide a full 64-bit counter.

counter is a pointer to the current value of the counter.

Return

The function always returns `RDIError_UnimplementedMessage`.

Usage

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

4.14.3 ARMul_AddCounterValue64

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

Syntax

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

where:

handle is no longer used.

arg1/*arg2* are the arguments passed to the `UnkRDIInfoUpcall()`.

counterval is the current value of the counter.

Return

The function always returns `RDIError_UnimplementedMessage`.

Usage

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

4.14.4 ARMul_BusRegisterPeripFunc

A peripheral model must call this function to register the peripheral with RVISS. This enables RVISS to call the model whenever it makes accesses to memory locations belonging to the peripheral.

Syntax

```
int ARMul_BusRegisterPeripFunc(enum BusRegAct act,
                               ARMul_BusPeripAccessRegistration *breg);
```

where:

act is the action you want. *act* must have one of the following values: `insert` or `remove`.

breg is a structure containing information for RVISS. You can obtain this structure by calling `ARMulif_ReadBusRange` (see *ARMulif_ReadBusRange* on page 4-85).

breg is a structure of type `ARMul_BusPeripAccessRegistration` (see *ARMul_BusPeripAccessRegistration* for details).

ARMul_BusPeripAccessRegistration

This structure and type are declared in the file `armul_bus.h`, in:

install_directory`\RVARMulator\ExtensionKit\...\...\`*platform*`\armulif`

In this path:

- *platform* is:
  - `win_32-pentium` for Windows
  - `linux-pentium` for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

The declaration is as follows:

```
typedef struct ARMul_BusPeripAccessRegistration {
    ARMul_BusPeripAccessFunc *access_func;
    void *access_handle;
    uint32 capabilities; /* See PeripAccessCapability_* below */
    struct ARMul_Bus *bus;
    /* 0=> normal peripheral, earlier in list than anything it
     * overlaps with. */
    unsigned priority;
     /* 0..100%
      * A higher number will be placed earlier in the list than
      * anything that it doesn't overlap with and has a lower access_frequency.
```

```
      */
    unsigned access_frequency;
    unsigned addr_size; /* Number of elements in range[] */
    AddressRange range[1];
} ARMul_BusPeripAccessRegistration;
```

where:

access_func Pointer to the function to call for a memory access in the given address range.

access_handle Pointer to object data for *access_func*.

capabilities See *PeripAccessCapability*.

bus This is returned by `ARMulif_QueryBus`. Do not alter it.

priority Use this field to assign a priority to peripherals. Zero is the highest priority. If peripherals have overlapping address ranges, the highest priority peripheral is accessed first. Lower priority peripherals are only accessed if higher priority peripherals return without processing the call.

access_frequency Use this field to inform RVISS which peripheral you expect to be accessed more frequently. This enables RVISS to access peripherals more efficiently. Assign the frequency as a percentage in the range 0% to 100%.

addr_size This is for future expansion. 1 is for 32-bit addresses. This is the only address size currently supported.

range The address range occupied by this peripheral.

PeripAccessCapability

This parameter defines the capabilities of the peripheral. It is the sum of the values of the individual capabilities (see Table 4-7 on page 4-81).

For example:

- A value of `0x20020` means that the peripheral can handle word data accesses, but not bytes, halfwords, or doublewords, and understands the **Endian** signal. This value is predefined as `PeripAccessCapability_Minimum`.

- A value of `0x20038` means that the peripheral can handle byte, halfword, and word data accesses, but not doubleword, and understands the **Endian** signal. This value is predefined as `PeripAccessCapability_Typical`.

Table 4-7 Peripheral access capabilities

Capability	Predefined name	Value
Byte	`PeripAccessCapability_Byte`	`0x8`
Half word	`PeripAccessCapability_HWord`	`0x10`
Word	`PeripAccessCapability_Word`	`0x20`
Doubleword	`PeripAccessCapability_DWord`	`0x40`
Peripheral accepts idle cycles	`PeripAccessCapability_Idles`	`0x10000` (**`unsigned long`**)
Peripheral understands **Endian** signal	`PeripAccessCapability_Endian`	`0x20000` (**`unsigned long`**)
Peripheral understands bytelanes	`PeripAccessCapability_Bytelane`	`0x40000` (**`unsigned long`**)

4.14.5 ARMulif_CoreCycles

This function returns, on core models that support it, the number of times the main pipeline has advanced.

Note

For ARM9 models this is a gated clock. The clock can be installed by the memory or interlocks.

Syntax

```
ARMTime ARMulif_CoreCycles(RDI_ModuleDesc *mdesc)
```

where:

mdesc is the handle for the core.

Return

Returns, on the core models that support it, the number of times the main pipeline has advanced.

4.14.6 ARMulif_CPUCycles

This function returns the time in units of CPU speed.

Note

Only supported on ARM10-based and XScale models.

Syntax

```
ARMTime ARMulif_CpuCycles(RDI_ModuleDesc *mdesc)
```

where:

mdesc is the handle for the core.

Return

Returns, on core models that support it, the time in units of CPUSPEED.

4.14.7 ARMulif_EndCondition

This function returns the *reason* passed to `ARMulif_StopExecution`.

Syntax

```
unsigned ARMulif_EndCondition(RDI_ModuleDesc *mdesc)
```

where:

mdesc is the handle for the core.

4.14.8 ARMulif_GetCoreClockFreq

This function returns the `CPUSPEED` in Hertz.

Syntax

```
ARMTime ARMulif_GetCoreClockFreq(RDI_ModuleDesc *mdesc)
```

where:

mdesc is the handle for the core.

4.14.9 ARMulif_InstallHourglass

Use this function to install an hourglass callback from RVISS to your model.

Syntax

```
void *ARMulif_InstallHourglass(RDI_ModuleDesc *mdesc,
                               armul_Hourglass *newHourglass, void *handle);
```

where:

mdesc is the handle for the core.

newHourglass is a function of type armul_Hourglass. See *ARMul_Hourglass* for more information.

handle is a pointer to the data required by your function, *newHourglass*.

Usage

When you install an hourglass, RVISS gives your model a callback each time an instruction is executed.

Return

This function returns a handle for your model to use to remove the hourglass callback.

ARMul_Hourglass

The prototype for armul_Hourglass is:

```
void armul_Hourglass(void *handle, ARMword pc, ARMword instr, ARMword cpsr,
                     ARMword condpassed)
```

where:

handle is the handle for the core.

pc is the address of the current instruction.

instr is the current instruction. For example, this is a 32-bit word for ARM instructions, or a 16-bit halfword for Thumb instructions.

cpsr is the current contents of the CPSR.

Note

This contains the mode bits, but does *not* reflect the correct contents of the flag bits.

condpassed is 0 if the condition of current instruction fails and the instruction is therefore not executed, or 1 otherwise.

Note

If your model uses this, it must test the bottom bit of *condpassed*. The use of the other bits is reserved.

4.14.10 ARMulif_ReadBusRange

You must supply a *breg* structure to register a peripheral. Call this function to initialize the fields in this structure.

Syntax

```
int ARMulif_ReadBusRange(struct RDI_ModuleDesc *mdesc,
                         struct RDI_HostosInterface const *hostif,
                         toolconf config,
                         struct ARMul_BusPeripAccessRegistration *breg,
                         uint32 default_base, uint32 default_size,
                         char const *default_bus_name);
```

where:

mdesc is the handle for the core.

hostif is the handle for the host interface.

config is the configuration passed in to your model in BEGIN_INIT.

breg is a structure containing information for RVISS. You require this for registerPeripFunc() (see *ARMul_BusRegisterPeripFunc* on page 4-79).

For details of the structure, see armulbus.h in:

install_directory\RVARMulator\ExtensionKit\...\...*platform*\armulif

In this path:

- *platform* is:
  - win_32-pentium for Windows
  - linux-pentium for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

default_base is the default base address to use for your peripheral. This address is used if *config* does not contain a base address for your peripheral.

default_size is the default size of the area in memory to use for your peripheral. This is used if *config* does not contain a size for your peripheral.

default_bus_name

is a pointer to a string. This string is used if no bus name is found in the config parameter for this peripheral, for example in a .dsc or .ami file.

4.14.11 ARMulif_RemoveHourglass

Use this function to remove an hourglass callback.

Syntax

```
int ARMulif_RemoveHourglass(RDI_ModuleDesc *mdesc, void *node);
```

where:

mdesc is the handle for the core.

node is the handle returned by `ARMulif_InstallHourglass`.

4.14.12 ARMulif_StopExecution

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

Syntax

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

where:

mdesc is the handle for the core.

reason is an `RDIError` error value. RealView Debugger interprets *reason* and issues a suitable message. Expected errors are:

`RDIError_NoError`
: Program ran to a natural termination.

`RDIError_BreakpointReached`
: Stop condition was a breakpoint.

`RDIError_WatchPointReached`
: Stop condition was a watchpoint.

`RDIError_UserInterrupt`
: Execution interrupted by the user.

4.14.13 ARMulif_Time

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

Syntax

```
ARMTime ARMulif_Time(RDI_ModuleDesc *mdesc)
```

where:

mdesc is the handle for the core.

Return

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

4.14.14 Hostif_RaiseError

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

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

`Hostif_RaiseError` must only be used during initialization.

Syntax

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

where:

hostif is the handle for the host interface.

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

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

4.15 Accessing the RealView Debugger

This section describes functions that you can use to access RealView Debugger.

Several functions are provided to display messages in the appropriate RealView Debugger window.

All the functions described in this section take the following as the first parameter:

```
const struct RDI_HostosInterface *hostif
```

This value is available in the state data structure of the model, as defined between the `BEGIN_STATE_DECL()` and `END_STATE_DECL()` macros (see *Basic model interface* on page 4-35).

4.15.1 Hostif_ConsolePrint

This function prints the text specified in the format string to the RVISS console. Under RealView Debugger, the text appears in the corresponding I/O window.

Syntax

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

where:

hostif is the handle for the host interface.

format is a pointer to a printf-style formatted output string.

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

Note

Use `Hostif_PrettyPrint()` to display startup messages.

4.15.2 Hostif_ConsoleRead

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

Syntax

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

where:

hostif is the handle for the host interface.

buffer is a pointer to a buffer to hold the string.

len is the maximum length of the buffer.

Return

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

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

4.15.3 Hostif_ConsoleReadC

This function reads a character from the RVISS console.

Syntax

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

where:

hostif is the handle for the host interface.

Return

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

4.15.4 Hostif_ConsoleWrite

This function writes a string to the RVISS console.

Syntax

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

where:

hostif is the handle for the host interface.

buffer is a pointer to a buffer holding a zero-terminated string.

len is the length of the buffer.

Return

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

4.15.5 Hostif_DebugPause

This function waits for the user to press any key.

Syntax

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

where:

hostif is the handle for the host interface.

4.15.6 Hostif_DebugPrint

This function displays a message in the RealView Debugger logging window or to the console when running RealView Debugger in command-line mode.

Syntax

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

where:

hostif is the handle for the host interface.

format is a pointer to a printf-style formatted output string.

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

4.15.7 Hostif_PrettyPrint

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

Syntax

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

where:

hostif is the handle for the host interface.

config is a pointer to the toolconf configuration database of the model. This value is available in the state data structure of the model, as defined between the `BEGIN_STATE_DECL()` and `END_STATE_DECL()` macros (see *Basic model interface* on page 4-35).

format is a pointer to a printf-style formatted output string.

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

4.15.8 Hostif_WriteC

This function writes a character to the RVISS console.

Syntax

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

where:

hostif is the handle for the host interface.

c is the character to write. *c* is converted to an unsigned char.

4.16 Tracer

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

Note

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

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

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

See also:

- *Tracer* on page 2-8.

4.16.1 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.

4.16.2 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.

4.16.3 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.

4.16.4 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.

4.17 Map files

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

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

See the *RealView Debugger User Guide* for details of how to use a map file in a debugging session.

To calculate the number of wait states for each possible type of memory access, RVISS uses the access times supplied in the map file, and the clock frequency from RealView Debugger (see the *RealView Debugger Target Configuration Guide*).

See also *Memory modeling with mapfiles* on page 2-29.

Note

A memory map file defines the characteristics of the memory areas defined in `peripherals.ami` (see *RVISS configuration files* on page 4-99). A `.map` file must define read/write areas that are at least as large as those specified for the heap and stack in `peripherals.ami`, and at the same locations. If this is not the case, Data Aborts are likely to occur during execution.

4.17.1 Format of a map file

The format of each line is:

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

where:

start The start address of the memory region in hexadecimal, for example `80000`.

size The size of the memory region in hexadecimal, for example, `4000`.

name 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 The width of the data bus in bytes (that is, 1 for an 8-bit bus, 2 for a 16-bit bus, or 4 for a 32-bit bus).

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

`r` for read-only.

`w` for write-only.

`rw` for read-write.

`-` for no access. Any access causes a Data or Prefetch Abort.

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

read-times

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

Note

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

write-times

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

The following examples assume a clock speed of 20MHz, the default.

Example 1

Example 4-2 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 4-2 Single contiguous section of RAM

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

Example 2

Example 4-3 describes a typical embedded system with 32KB of on-chip memory, 16-bit ROM and 32KB of external DRAM:

Example 4-3 Embedded system memory map

```
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
```

The regions of memory are:

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

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

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

Note

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

To ensure that you have described all areas of memory that you think the image accesses, you can define a single memory region that covers the entire address range as the last line of the map file. For example, you can add the following line to Example 4-3:

```
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 by examining the values of the `@mapfile_`*`symbolname`* symbols.

Note

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

Displaying the memory statistics

To get a list of the `mapfile` symbols in RealView Debugger use the `REGINFO` command as follows:

```
reginfo,access,match:@mapfile
```

To display the memory statistics use the RealView Debugger `PRINTF` command, for example:

```
printf "%d", @mapfile_symbolname
```

See also

- *RealView Debugger User Guide*
- *RealView Debugger Target Configuration Guide*
- *RealView Debugger Command Line Reference Guide* for details of the `PRINTF` and `REGINFO` commands.

4.18 RVISS configuration files

RVISS configuration files (`.ami` and `.dsc` files) are ToolConf files, which are located in the following directories:

- The main RVISS configuration directory contains `.ami` and `.dsc` files:

  install_directory`\RVARMulator\ARMulator\...\...\`*platform*

  By default, the following `.ami` files are supplied:
  - `bustypes.ami`
  - `default.ami`
  - `example1.ami`
  - `peripherals.ami`
  - `processors.ami`
  - `vfp.ami.`

  By default, the following `.dsc` files are supplied:
  - `arm925.dsc`
  - `arm1020ej.dsc`
  - `arm1026ej.dsc`
  - `armiss.dsc`
  - `armulate.dsc`
  - `peripherals.dsc`
  - `v6armiss.dsc`
  - `vfp11.dsc`
  - `vfp.dsc`
  - `xScale.dsc.`

- For the MPCore™ model, the following directory contains the `smp11.dsc` file:

  install_directory`\RVARMulator\MPCore\ARMulator\...\...\rvds30\`*platform*

- For ARMv6 architecture models that support Thumb-2 and TrustZone™, the following directory contains the `v6thumb2.dsc` and `v6trustzone.dsc` files:

  install_directory`\RVARMulator\v6ARMulator\...\...\`*platform*.

In these paths:

- *platform* is:
  - `win_32-pentium` for Windows
  - `linux-pentium` for Red Hat Linux.
- For Red Hat Linux, replace \ with /.

When you connect to an RVISS model from RealView Debugger, RVISS loads all `.ami` files it finds on any of the paths specified in the environment variable `ARMCONF`. This is initially set up to point to:

install_directory\RVARMulator\ARMulator\...\...*platform*

If a configuration is specified differently in two files, the file in the path that appears first is used. If there are several directories in `ARMCONF`, RVISS loads `.ami` files from directories in the order that they appear in the list. RVISS loads `.ami` files from within each directory in an unpredictable order.

4.18.1 Predefined tags

Before reading `.ami` files, RVISS creates several tags itself, based on the settings you give to RealView Debugger. These are given in Table 4-8. Preprocessing directives in `.ami` files use these tags to control the configuration.

Table 4-8 Tags predefined by RVISS

Tag	Description
`CPUSpeed`	Set to the speed set in the RealView Debugger ARMulator Configuration dialog box, or in the `-clock` command line option when running RealView Debugger in command-line mode. For example, `CPUSpeed=30MHz`.
`FCLK`	Set to the same value as `CPUSpeed`, if that value is not zero. Not set if `CPUSpeed` is zero.
`MCLK`	Set to the same value as `FCLK` for uncached cores. Set to `FCLK/MCCFG` for cached cores.
`ByteSex`	Set to `L` or `B` if a bytesex is specified from RealView Debugger. Not set otherwise.
`FPE`	Set to `True` or `False` from RealView Debugger.

4.18.2 Processors

The processors region is a child ToolConf database (see *ToolConf* on page 4-105). It has a full list of models supported by RVISS. This list is the basis of:

- the list of processors in the RealView Debugger ARMulator Configuration dialog box
- the list of accepted targets for the `--target` option when starting RealView Debugger from the command-line.

You can add a variant processor to this list, for example to include a particular memory model in the definition. For examples, see the `example1.ami` file in:

install_directory\RVARMulator\ARMulator\...\...*platform*

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

Example 4-4 declares two processors, `TRACED_ARM10` and `MAPPED_ARM7`. In this example, `MCCFG` is the ratio of the clock frequency on the processor to the clock frequency on the external bus.

Example 4-4 Processors in a toolconf file

```
{PROCESSORS

  {TRACED_ARM10=ARM10200E

    ;CPUSPEED=400MHz

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

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

  ;End TRACED_ARM10
  }

  {MAPPED_ARM7=ARM720T
```

```
    {Flatmem
      {Peripherals
        {Mapfile=Default_Mapfile
          MAPFILETOLOAD=C:\Myprojects\arm7_map.map
        }
      }
    }
  ;End MAPPED_ARM7
  }

;End Processors
}
```

Finding the configuration for a selected processor

RVISS 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.

See also

- *ToolConf* on page 4-105
- *RealView Debugger User Guide*
- *RealView Debugger Target Configuration Guide*
- *RealView Debugger Command Line Reference Guide*.

4.18.3 Adding a variant processor model

Suppose you have created a memory model called `MyASIC`, designed to be combined with an ARM7TDMI processor core to make a new microcontroller called `ARM7TASIC`.

To enable your variant processor model to be selected from RealView Debugger:

1. Create a new `.ami` file modeled on `example1.ami`.
2. Add your variant processor model to the following RealView Debugger configuration file:

   `$RVDEBUG_INSTALL\etc\armul.var`

4.18.4 Changing the cache or TCM size of a synthesizable processor

To change the cache or TCM size of a synthesizable processor:

1. Create a new `.ami` file based on the `processors.ami` file containing:

```
{PROCESSORS
  {Variant_name=processor_name
    configuration_settings
  }
}
;End of Processors

{UNIREGSNAMES
Variant_name=processor_name
}
```

2. Place your new `.ami` file in the same location as the `processors.ami` file (see *RVISS configuration files* on page 4-99).

For example, to change both the instruction and data caches of an ARM946E-S to 8KB:

1. Edit your new `.ami` file.
2. In the `PROCESSORS` section enter the configuration settings for your variant:

```
{PROCESSORS
  {ARM946E-S_Cached=ARM946E-S
    ICache_Lines=256
    DCache_lines=256
  }
}
```

3. In the `UNIREGSNAMES` section, add an entry that corresponds to the configuration in the `PROCESSORS` section:

```
{UNIREGSNAMES
ARM946E-S_Cached=ARM946E-S
}
```

This overrides the corresponding lines in armulate.dsc.

Caution

Any processors that inherit properties from ARM946E-S are also affected if you make this change.

Cores that do not inherit their properties from ARM946E-S, such as ARM946E-S-REV0 or ARM946E-S-REV1, are not affected.

If you want to change the cache or TCM size of a processor that does not already have a section in the `.ami` file containing your variants, you can add a section. For example, to change the instruction RAM size of the ARM926EJ-S from 64KB to 32KB:

1. Edit the `.ami` file containing your variants.
2. Insert the following entry in the `PROCESSORS` section:

```
{ARM926EJ-S=Processors_Common_ARMULATE
  IRamSize=0x8000
}
```

This overrides the corresponding line in armulate.dsc.

Any details that are not specified in your file remain unaltered from those specified in `armulate.dsc`.

4.19 ToolConf

This section describes the RVISS ToolConf module

4.19.1 Toolconf overview

ToolConf is a module within RVISS. A ToolConf file is a tree-structured database consisting of tag and value pairs. Tags and values are strings, and are usually case-insensitive. ToolConf files are files of type `.ami` or `.dsc`.

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

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

4.19.2 File format

The following are typical ToolConf database lines:

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

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

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

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

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

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

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

A comment must be on a line by itself.

File header

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

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

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

Tree structure

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

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

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

Specifying children

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

One is more suited to specifying large children:

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

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

The other is more suited to specifying small children:

```
TagP:TagC=ValueC
```

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

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

Conditional expressions

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

```
#define Control True

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

maps to the ToolConf sequence:

```
Control=True

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

A condition is evaluated from left to right, on the contents of the configuration at that point. Table 4-9 shows the operators that can be used in ToolConf conditional expressions.

Table 4-9 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.

4.19.3 Boolean flags in a ToolConf database

Table 4-10 shows the full set of permissible values for Boolean flags. The strings are case-insensitive.

Table 4-10 Boolean values

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

4.19.4 SI units in a ToolConf database

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

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

The scaling factor is set by the prefix to the unit. RVISS 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}. RVISS decides which scaling to use according to context.

4.19.5 ToolConf_Lookup

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

Syntax

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

where:

hashv is the database to perform the lookup on.

tag is the tag to search for in the database. The tag is case-dependent.

Return

The function returns:

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

Example

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

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

4.19.6 ToolConf_Cmp

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

Syntax

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

where:

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

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

Return

The function returns:

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

Example

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

4.20 Reference peripherals

This section describes the following reference peripherals:

- interrupt controller
- timer.

4.20.1 Interrupt controller

The base address of the interrupt controller, `IntBase`, is configurable (see *Interrupt controller* on page 2-34).

Table 4-11 shows the location of individual registers.

Table 4-11 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 4-12 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 4-12 Interrupt sources

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

Note

Timer 1 and Timer 2 can be configured to use different bits in the IRQ controller registers, see *Timer* on page 2-35.

4.20.2 Timer

The base address of the timer, `TimerBase`, is configurable (see *Timer* on page 2-35).

See Table 4-13 for the location of individual registers.

Table 4-13 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 4-15 and Table 4-14 for details of timer register bits. Only bits 7, 6, 3, and 2 are used. You must write all others as zeroes.

Table 4-14 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 4-15 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.

Appendix A
Using MPCore Models

This appendix describes The MPCore™ model. It contains the following sections:

A.1 About MPCore

The MPCore system from ARM is an ARM11-based *Symmetric Multiprocessor* (SMP) solution available with between 1 and 4 cores. It is supplied with a set of peripherals including a Snoop Control Unit to ensure cache coherency between the cores, Timer/Watchdog units and an Interrupt Distributor. The MPCore model is single-core.

Available models are:

- `MPCore_x1_RVDS`, with 32K each of D-cache and I-cache
- `MPCore_x1_rvds_16k`, with 16K each of D-cache and I-cache
- `MPCore_x1_rvds_32k`, with 32K each of D-cache and I-cache (this is the same as `MPCore_x1_RVDS`)
- `MPCore_x1_rvds_64k`, with 64K each of D-cache and I-cache.

A.2 Default peripheral system

The ARM MPCore specification gives a register map relative to the parameter PERIPHBASE, which for the MPCore model is 0x1F00_0000. This allows a core to address its local peripherals through a fixed offset (independent of core number) from PERIPHBASE. In addition, these registers are aliased in the global memory maps so that other cores can access them.

The following table shows the memory map for the MPCore model. All regions not shown here contain RAM and are shared by all cores.

Table A-1 MPCore memory map

Physical address range	Registers
0x1F00_0000 to 0x1F00_00FF	Snoop Control Unit's registers
0x1F00_0100 to 0x1F00_01FF	Interrupt distributor local to current core
0x1F00_0200 to 0x1F00_02FF	Reserved for P0 in multi-core systems
0x1F00_0300 to 0x1F00_03FF	Reserved for P1 in multi-core systems
0x1F00_0400 to 0x1F00_04FF	Reserved for P2 in multi-core systems
0x1F00_0500 to 0x1F00_05FF	Reserved for P3 in multi-core systems
0x1F00_0600 to 0x1F00_06FF	Timer/watchdog local to current core
0x1F00_0700 to 0x1F00_07FF	Reserved for P0 in multi-core systems
0x1F00_0800 to 0x1F00_08FF	Reserved for P1 in multi-core systems
0x1F00_0900 to 0x1F00_09FF	Reserved for P2 in multi-core systems
0x1F00_0A00 to 0x1F00_0BFF	Reserved for P3 in multi-core systems
0x1F00_1000 to 0x1F00_1FFF	Global interrupt distributor

The following table shows the configuration of the interrupt distributor in MPCore.

Table A-2 Interrupt distributor configuration

Pin	Source	Signal	Banked?
29p00	TWD0	Timer IRQ	Yes
30p00	TWD0	Watchdog IRQ	Yes
29p01	TWD1	Timer IRQ	Yes

Table A-2 Interrupt distributor configuration (continued)

Pin	Source	Signal	Banked?
30p01	TWD1	Watchdog IRQ	Yes
29p02	TWD2	Timer IRQ	Yes
30p02	TWD2	Watchdog IRQ	Yes
29p03	TWD3	Timer IRQ	Yes
30p03	TWD3	Watchdog IRQ	Yes
51	P0	Performance Monitoring Unit IRQ	No
54	P1	Performance Monitoring Unit IRQ	No
57	P2	Performance Monitoring Unit IRQ	No
60	P3	Performance Monitoring Unit IRQ	No
63	SCU	Snoop Control Unit IRQ (overflow from MN0)	No
64	SCU	Snoop Control Unit IRQ (overflow from MN1)	No
65	SCU	Snoop Control Unit IRQ (overflow from MN2)	No
66	SCU	Snoop Control Unit IRQ (overflow from MN3)	No
67	SCU	Snoop Control Unit IRQ (overflow from MN4)	No

A.3 Limitations

The MPCore model has the following limitations:

- the Snoop Control Unit has a configuration register that reports the tag RAM sizes for the different cores in the system. The real hardware will report non-existing cores as having 0 as the tag RAM configuration for those cores. In the model, it will always be reported the same as for the core that exists.
- The Snoop Control Unit is only modelled functionally.
- The model is an instruction set simulator and does not have accurate timing. The model must therefore not be used for benchmarking.
- You cannot add new peripherals in the same way that you can with the MPCore processor system. However, for the single core version of the MPCore model, you can add old-style RVISS model peripherals. See *Writing a new MPCore model* on page A-6.
- The VA to PA register block is not implemented. These registers will produce errors if you attempt to access them. They are also inaccessible from real code.
- The MPCore processor is available with or without a VFP. The model always has a VFP.

A.4 Writing a new MPCore model

This section describes how to create your own MPCore model.

Note

The MPCore model in RVISS is a single-processor model.

A.4.1 Interrupts

The MPCore model already has an interrupt distributor (see *Default peripheral system* on page A-3). You therefore do not need to include one. RVISS peripherals that generate interrupts to a *Vectored Interrupt Controller* (VIC) are automatically redirected to the interrupt distributor that is part of the MPCore system.

Note

An MPCore system is not likely to contain a VIC and this is therefore not expected to occur in real hardware. The model is set up in this way for convenience. Input pins `0..18` of the VIC are mapped to pins `32..50` of the interrupt distributor.

When a peripheral generates an interrupt it uses code similar to the following:

```
....
GenericAccessCallback **interrupt_controller;
...

void change_interrupt_pin( MyState* state, unsigned pin, unsigned level )
{
    if (state->interrupt_controller == NULL) {
        state->interrupt_controller = ARMulif_GetInterruptController(
            &state->coredesc
            );
        if (state->interrupt_controller == NULL) {
            /* ... error ... */
            return;
        }
    }

    {
        GenericAccessCallback* ic = *state->interrupt_controller;
        if (ic != NULL) {
            ic->func(ic, pin, &level, 0);
        }
        else {
            /* ... error ... */
            return;
```

```
        }
    }
}
```

If `pin` is within the assigned range (`0..18`) then it is redirected to the interrupt distributor. The exact pin that each peripheral communicates with is dependent on the peripheral and the way in which it is configured.

Peripherals that do not use the VIC interface can signal the core directly by using the **IRQ/FIQ** signal, but this is not recommended. The **IRQ/FIQ** signal does not interact with the interrupt distributor, and the interrupt service routine must independently locate the source of the interrupt. It is recommended that you modify these peripherals to map to the interrupt distributor.

A.4.2 Configuring your new model

The MPCore model does not automatically load the peripherals in `PeripheralSets:Default_Common_Peripherals`, but instead uses its own section `PeripheralSets:RVISSStylePeripherals`. This means that any peripherals that are incompatible with the `MPCore_x1_RVDS` model can be loaded unchanged into other RVISS models.

`PERIPHBASE` is the base address of the memory-mapped MPCore peripheral registers. `PERIPHBASE` is set to `0x1F00_0000`.You cannot change this setting.

A.4.3 Setting up your new model

It is possible to setup different versions of an MPCore system with a customized peripheral set. Use the following to set up your new model:

```
;; ARMulator configuration file type 3

; This file must have the suffix '.ami'

{ Processors
  { MyMpCoreSystem = MPCore_x1_rvds
    ;or MPCore_x1_rvds_16k, MPCore_x1_rvds_32k or MPCore_x1_rvds_64k,
    ;as required    PERIPHERAL_SYSTEM=MyPeripherals
  }
}
{ PeripheralSets
  { MyPeripherals=RVISSStylePeripherals_MustHaves
    { MyPeripheral      ...
    }
  }
}
{ UniregsNames
```

```
  ; tell the debugger what core is in the model systemMyMpCoreSystem =
MPCore,VFPv2
}
```

See the *RealView Debugger Target Configuration Guide* for information on how to select your new model.

Appendix B
ARM1136JF-S and ARM1136J-S Models

This appendix describes the ARM1136JF-S™ and ARM1136J-S™ models. It contains the following section:

- *Restrictions for the ARM1136JF-S and ARM1136J-S models* on page B-2.

B.1 Restrictions for the ARM1136JF-S and ARM1136J-S models

The following restrictions apply to ARM1136JF-S and ARM1136J-S models:

- The caches are not modelled accurately enough to show aliasing effects for cache sizes greater than 16kB.
- Under RealView Debugger the Invalidate by Range operations that are new to the ARMv6 architecture are shown as 64 bit registers. The high 32 bits of this register is the start address and the low 32 bits are the end address.
- Table B-1 shows the registers that are not modeled for the ARM1136JF-S and ARM1136J-S models.

Table B-1 Registers not modeled for ARM1136JF-S and ARM1136J-S RVISS models

Register	RealView Debugger symbol
Data Memory Remap Register	`@CP15_D_MEM_REMAP`
Instruction Memory Remap Register	`@CP15_I_MEM_REMAP`
DMA Memory Remap Register	`@CP15_DMA_MEM_REMAP`
Peripheral Port Memory Remap Register	`@CP15_PERPH_MEM_REMAP`
Instruction Cache Master Valid Register	`@CP15_ICACHE_MASTER_VALID_REGISTER_0/7`
Instruction SmartCache Master Valid Register	`@CP15_ISMARTCACHE_MASTER_VALID_REGISTER_0/7`
Data Cache Master Valid Register	`@CP15_DCACHE_MASTER_VALID_REGISTER_0/7`
Data SmartCache Master Valid Register	`@CP15_DSMARTCACHE_MASTER_VALID_REGISTER_0/7`
Main TLB VA Register	`@CP15_TLB_MAIN_VA`
Main TLB PA Register	`@CP15_TLB_MAIN_PA`
Main TLB Attribute Register	`@CP15_TLB_MAIN_ATT`