

Application Note **32**

The ARMulator

Document number: ARM DAI 0032F

Issued: September 2003

Copyright ARM Limited 2003

ARM

Application Note 32
The ARMulator

Copyright © 1996-2003 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
August 1996	A	First release
August 1996	B	Changes to remove references to integration with hardware modeling environments.
January 1998	C	Changes to incorporate new models introduced in SDT version 2.10 and 2.11
April 1999	D	Changes to reflect SDT 2.50.
August 2001	E	Changes to reflect ADS 1.1.
September 2003	F	Changes to reflect ADS 1.2, RVD 1.6.1, RVARMulator ISS 1.3, 1.3.1, and RVCT 2.0.1

Proprietary notice

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited. The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, ARM7TDMI, ARM9TDMI, TDMI, STRONG and RealView are trademarks of ARM Limited. All other products, or services, mentioned herein may be trademarks of their respective owners.

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to support@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

<http://www.arm.com>

Table of Contents

Table of Contents	3
1 Introduction.....	4
1.1 The ARMulator Extension Kit.....	4
1.2 RealView ARMulator Instruction Set Simulator	5
1.3 Cached versus Uncached Targets	5
2 The Structure of the ARMulator	6
2.1 The ARM processor core model	6
2.2 The memory system.....	7
2.3 The coprocessor interface	7
2.4 The operating system interface.....	7
3 Modeling Systems Using the ARMulator	9
3.1 Modifying ARMulator	9
3.2 Generating exceptions.....	12
3.3 Event scheduling.....	12
4 Example: Parallel Port Model	15
4.1 Creating the peripheral model (Windows procedure).....	15
4.2 Explanation	16
4.3 ARM application code	17
4.4 Running the application	17
4.5 Running the Example on RealView Debugger	18
5 Example: Exception Generator Memory Model	19
5.1 Creating and modifying the files.....	19
5.2 Writing code to access the memory locations	19
5.3 Writing ARM application code.....	19
5.4 Running the application	19
6 Example: Coprocessor Model	21
6.1 Creating the files	21
6.2 Editing files.....	21
6.3 Writing application files	21
6.4 Running the code	21
7 Debugging ARMulator models in Visual C++	22
7.1 Creating a project.....	22
7.2 Adding files	22
7.3 Configure project settings.....	22
7.4 Compile the module	23
7.5 Ensure .dsc and .ami configuration files have been properly configured.....	23
7.6 Set breakpoints	23
7.7 Launch debugger	23
8 Calling a Peripheral Every Cycle	24
9 Communication between ARMulator models	28
9.1 How to model shared memory.....	28
9.2 How to coordinate execution	29
10 Known Issues.....	30

1 Introduction

The ARMulator is a family of programs which emulate the instruction sets of various ARM processors and their supporting architectures.

The ARMulator:

- provides an environment for the development of ARM-targeted software on a range of non-ARM-based host systems
- allows benchmarking of ARM-targeted software (though its performance is somewhat slow compared to real hardware)¹
- supports the simulation of prototype ARM-based systems, ahead of the availability of real hardware, so that software and hardware development can proceed in parallel.

The ARMulator is transparently connected to compatible ARM debuggers to provide a hardware-independent ARM software development environment. Communication takes place via the Remote Debug Interface (RDI)².

This document describes the structure and use of ARMulator, as well as providing a number of examples to help you to develop your own models. The source files and pre-built .dlls and images can be downloaded from http://www.arm.com/arm/Application_Notes.

For further details, refer to the *ADS 1.2 Debug Target Guide* (ARM DUI 0058D) or the *RealView ARMulator ISS 1.3 User Guide* (ARM DUI 0207A).

Notes This Application Note is designed for ARM Developer Suite 1.2 running on Windows. You will need to generate your own makefiles for use on Solaris, Linux or HP-UX (if applicable), using the supplied Windows makefile as a guide. For information on extending the ARMulator for SDT or ADS 1.0 and 1.1 please see previous revisions of Application Note 32. RealView Debugger interacts with ARMulator in a different way which means that not all features discussed here may be available. These are noted in the text.

1.1 The ARMulator Extension Kit

The ARMulator extension kit contains source code for a selection of models. You can modify these models to generate new peripherals, memory maps and coprocessors, or interact with the operating system via software interrupts. The ARMulator extension kit is included if you have made a full installation, or a custom installation which includes the extension kit. The location of the kit varies, depending on which ARMulator you are using, and what platform you are working with.

If you are using the ARM Developer Suite on Windows, the extension kit is located in

`<install_path>\ARMulate\armulext`

If you are using ADS on Unix or Linux, the extension kit can be found in

`<install_path>/<platform>/Source/armulext/`

where `<platform>` is one of `solaris`, `hpux` or `linux`.

¹ In general, models of the less complex, uncached ARM processor cores are cycle accurate, but models of the cached variants might not correspond exactly with the actual hardware.

² All ARM debuggers including RealView Debugger 1.6.1 on Windows support the RDI protocol, though RVD's RDI support is limited. The RDI interface is not available in RVD on Unix, nor is it expected to be supported in future releases of RVD.

If you are using RealView ARMulator ISS, the extension kit is in a location similar to

```
<install_path>/ARM/RVARMulator/ExtensionKit/1.3/release/  
<platform>/armulext
```

where <platform> is one of win_32-pentium, solaris-sparc, or linux-pentium. On Windows, the / should be replaced by \.

1.2 RealView ARMulator Instruction Set Simulator

This document can be used with RealView ARMulator Instruction Set Simulator version 1.3 and 1.3.1.

The model source directory is located within the ExtensionKit subdirectory of RVARMulator. You will need to follow this directory structure down until you reach the win_32-pentium, solaris-sparc or linux-pentium directory, which contains the armulext source appropriate for your platform. The source directory is present if you have done a full installation, or a custom installation including the Extension Kit.

After you have built your model you will need to move it to the RVARMulator executables directory. This is normally a location similar to:

```
C:\Program Files\ARM\RVARMulator\ARMulator\1.3.1\1310.20030312\win_32-  
pentium
```

or

```
<$ARMROOT>/RVARMulator/ARMulator/1.3.1/1309.20030208/solaris-sparc
```

or

```
<$ARMROOT>/RVARMulator/ARMulator/1.3.1/1309.20030208/linux-pentium
```

Note Because of known issues with semihosting, you should not develop ARMulator models for RealView Debugger 1.6.1 using the LocalHost connection on WindowsNT.

1.3 Cached versus Uncached Targets

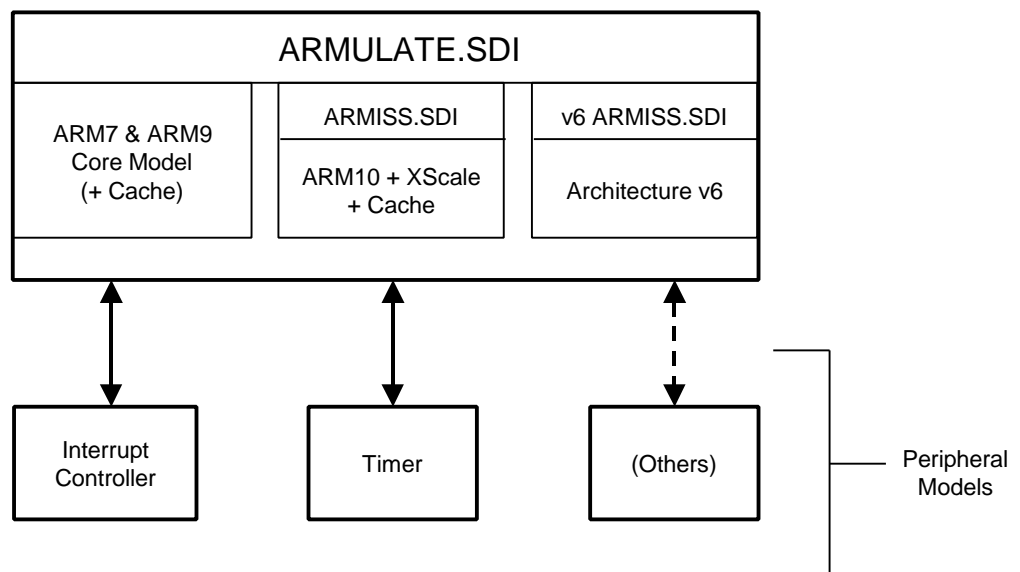
The examples supplied with this Application Note will work on uncached ARMulator targets such as the ARM7TDMI. However, even without pagetables enabled, the examples may not work properly on cached targets.

2 The Structure of the ARMulator

The ARMulator comprises several parts:

- a model of the ARM processor core and cache (if used)
- peripheral models that communicate with the base memory model and may be enabled or disabled via configuration files
- an operating system interface to provide an execution environment.

By modifying or rewriting the supplied models, you can model almost any ARM system and use it to debug code. The following diagram illustrates this structure.



Peripherals are registered by calling the functions `ARMulif_ReadBusRange` and `bus_registerPeripFunc` during model initialisation. This is explained fully with an example in section 4.2 Explanation (Parallel Port Model). Address settings may either be hard-coded or loaded in from a configuration (`.ami` or `.dsc`) file prior to registration. The user provides:

- a base address at which the peripheral is located
- the number of bytes which are covered by the peripheral model.

It is possible to have gaps within this range which are not decoded by a peripheral.

In the diagram above, `ARMULATE.SDI` represents the main ARMulator component. Below this resides a model of the core being emulated along with any cache if it has been configured. At the lowest level is a flat memory model (with the full 32 bit, 4GB range accessible) and the peripheral decoder. Prior to ADS 1.2, the decoder and flat memory model were integral to the ARMulator and could not be modified. However, this is not the case in ADS 1.2 and RV ARMulator ISS.

2.1 The ARM processor core model

The ARM processor core model handles all communication with the debugger. This part of the ARMulator is not customizable.

2.2 The memory system

The memory interface transfers data between the ARM model and the memory model or memory management unit model.

The memory model is fully customizable. Sample implementations are provided with the ARMulator. You can define features such as models of peripheral registers, memory mapped I/O, trigger regions for external interrupts, DMA models and so on, by modifying the memory model.

The default memory model is 4GB of zero-wait state RAM. The default memory model is used if you do not specify a mapfile in AXD, ADU, or ADW.

Mapfile (in `mapfile.c`) 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.

Please refer to the *Debug Target Guide* or *RV ARMulator ISS User Guide* sections 2.6 and 2.7 for more information.

You may add additional functions between the `BEGIN_INIT` and `END_INIT` macros in your memory model to perform any startup time initialization of your memory system extensions. Similarly, you may add additional functions between the `BEGIN_EXIT` and `END_EXIT` macros to free up any dynamically allocated memory.

A structure providing access to the state of the memory system model is declared using the `BEGIN_STATE_DECL` and `END_STATE_DECL` macros. You may add private data used by your model between the two macros.

Please refer to the *Debug Target Guide* or *RV ARMulator ISS User Guide* section 3.2.3 for more information.

2.2.1 Simple memory modeling

For modeling memory systems with different RAM types and access speeds, the standard ARMulator model supports memory map files. You can also use your own memory models to support map files by using `mapfile.c` as a template.

The map file defines areas within, and access speeds to, the emulated memory accessed by the emulated ARM. This is used to assist in calculating the performance of the ARM at the given clock speed with the given memory map. It does not control how the emulated memory relates to the host's real memory.

The use of map files is explained in detail in the *Debug Target Guide* or *RV ARMulator ISS User Guide*, section 4.13, Map files.

Note ARMulator mapfiles are not supported by RealView Debugger.

2.3 The coprocessor interface

The coprocessor model is called whenever the ARM executes a coprocessor instruction. The model can be used to simulate attached ARM-style coprocessors (such as floating point accelerators or custom DSPs).

The process for adding a coprocessor model is described in section 3.1.3, Adding a coprocessor model.

2.4 The operating system interface

The operating system model is called whenever the ARM executes a software interrupt (SWI) instruction, so you can simulate the operating system (or debug monitor) in C without having to write any ARM code.

The semihosting nature of the ANSI C library means that many of the C functions, such as file I/O, are implemented on the host computer, via the host's C library. These host services are accessed using SWI calls to the debug monitor (see Chapter 5 in the *Debug Target Guide*³).

The operating system model directly implements some operating system calls (such as open file, read the clock and so on) on the debugger host. These calls form the basis for the library calls (for example, `fopen()` and `time()`) provided by the ANSI C library.

This part of the ARMulator is also fully customizable. You can add extra SWIs to provide more host system functionality to the user. SWIs that are not handled by this model take the SWI trap and can be handled by ARM SWI handler code running on the ARMulator.

If you have an embedded system where SWIs are not used, you can remove the operating system entirely.

Refer to the *Debug Target Guide* for a full description of the Application Programming Interface (API) between the ARM debugger and the memory model, coprocessor model and operating system.

³ The *RV ARMulator ISS User Guide* does not document the semihosting mechanism. You will need to refer to the *ADS Debug Target Guide*, or chapter 7 of the *RealView Compilation Tools Compilers and Libraries Guide* (DUI 0205B). Both documents are downloadable from the ARM website at http://www.arm.com/arm/User_Guides.

3 Modeling Systems Using the ARMulator

You can make a model of almost any ARM system by modifying or rewriting the ARMulator default models. Before you can use a new model, you need to rebuild it as explained below.

3.1 Modifying ARMulator

Depending on your needs, there are a number of approaches that you can take to modify the ARMulator memory system.

There are three main types of model which may be implemented:

- Memory or Peripheral model
 - An entirely new memory model may be derived from `armmap.c`. This model may therefore make use of memory timing specifications taken from a supplied map file.
 - A peripheral or other memory-mapped device can be assigned an address range within the 4GB address space of the ARM core. Such devices are loaded after the ARMulator core and requests within a peripherals range are redirected to the appropriate module. It is possible to model a complete memory system by mapping a model to the full address space.
- Coprocessor model
 - Each coprocessor may be assigned to one of the 16 coprocessor numbers. This enables the basic instruction set to be expanded, to perform floating point operations for example.
- Operating System Interface model
 - Input/output requests may be communicated from application code to a host computer running a debugger. This is achieved by defining Software Interrupt handlers to respond from SWIs generated by your application.

Refer to the *Debug Target Guide* or *RV ARMulator ISS User Guide*, Chapter 3, Writing ARMulator models, for details of how to build a model.

3.1.1 Editing a copy of existing files

The simplest arrangement is to make a working copy of the rebuild kit and take copies of the files for the example model which most closely matches your intended design. For a complete listing of the example modules refer to the *Debug Target Guide* or *RV ARMulator ISS User Guide*, section 3.1.2 Supplied Models.

The model contained in `nothing.c` performs no useful operations but can be used to disable unused ARMulator models in a configuration file. It is also a useful template for building models from scratch. Refer to the *Debug Target Guide* or *RV ARMulator ISS User Guide*, section 3.5 Configuring ARMulator to disable a model.

3.1.2 Adding a memory or peripheral model

Unlike earlier versions, the ARMulator supplied with ADS 1.1 or 1.2, or RealView ARMulator ISS, does not need to be recompiled whenever a new model is added. Each model is contained within a standalone library and may be loaded by adding an entry to one or more configuration files as explained below.

The procedure is as follows:

- create new models
- copy an existing makefile directory for use with the new model
- create a `.dsc` file
- make changes to `default.ami` and `peripherals.ami`
- build the new model
- copy the resulting library into the correct directory.

The details are given on the following pages.

To add a new model to ARMulator

This section outlines the procedure for building and adding a new model. You can also use Visual C++ on Windows to build your model: see section 7 for more details.

- 1 If you have not already done so, make a working copy of the ARMulator extension kit (included in a full or custom installation). Make your changes to the working copy, *not* the original files.
- 2 Place new sources for your memory or peripheral model in the ARMulator extension kit directory. See sections 1.1 and 1.2 for where this directory can be found.
- 3 Create a new copy of one of the existing directories (named `<MODEL_NAME>.b`) within the source directory and rename it to reflect your model name (for example, `MyModel`).
- 4 Edit the Makefile inside the new directory's platform subdirectory, replacing all occurrences of `<MODEL_NAME>` with your new model name, `MyModel`. This makefile can be found in: `<extension kit>/MyModel.b/<platform>` where `<platform>` is one of `intelrel`, `gccsolrs`, `cchppa`, `linux86`. See sections 1.1 and 1.2 for the value of the extension kit directory for your system. On Windows, the `/` should be replaced by `\`.
- 5 Change your current directory to the one where your makefile is found. See step 4.
- 6 Type the name of your `make` utility. Note that only gcc version 2.95.2 is supported.
- 7 The built `MyModel.dll` (or `MyModel.so` or `MyModel.sl`, depending on your platform) should appear in the same directory as your makefile. See step 4. If you are using ADS move `MyModel.dll` (or `MyModel.so` or `MyModel.sl`) to: `<install_path>\bin` on Windows, or `<install_path>/<platform>/bin`, where `<platform>` is one of `solaris`, `hpux` or `linux`. If you are using RVARmulator ISS, move the file to the location given in section 1.2. This is where ARMulator expects to find models.

To run ARMulator with the new memory model

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

- 1 Create a file called `MyModel.dsc` and place it in the same directory as the `.dll` (or `.so` or `.sl`). The `.dsc` file must contain the following and be formatted as below:

```

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

```

If you are using ADS 1.1, add a `META_SORDI_DLL=MyModel` entry after the `MODEL_DLL_FILENAME=MyModel` entry.

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

```

{Tracer=Default_Tracer
}

```

- 3 Add the reference to your model:

```

{Tracer=Default_Tracer
}

{MyModel=Default_MyModel
}

```

- 4 Save your edited `default.ami` file.

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

```

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

```

Using this as an example, add a configuration section for your model. User-editable settings are typically stored in an `.ami` file. ARMulator will load any `.ami` or `.dsc` files located in the path given by the `ARMCONF` environment variable. Refer to the *Debug Target Guide* or *RV ARMulator ISS User Guide*, section 4.15.2 File format for details of how to construct configuration files.

The base address and size of the peripheral in the memory map, if required, is determined by an `ARMulif_ReadBusRange` call in `MyModel.c`. The first value is the base address, the second value is the size. The settings in `peripherals.ami` will override the base settings in the model `ARMulif_ReadBusRange` call if different. See the supplied model source code for examples.

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

3.1.3 Adding a coprocessor model

You may add extra coprocessor models by using the same procedure as described in the previous section. `MyModel` is replaced with the name of the coprocessor model. Coprocessor models differ from the above in the callbacks which are supported and the way in which they register themselves with the ARMulator. An example coprocessor model is presented in section 6.

3.2 Generating exceptions

When modeling a target system for code development, it is often necessary to be able to generate exceptions, such as IRQ, FIQ and data aborts. This section deals with the immediate generation of exceptions. The next section describes a means of scheduling the generation of events, such as exceptions, some number of cycles into the future.

3.2.1 IRQ

Provided that the CPSR I bit (bit 7) is 0, an IRQ may be generated by calling the function

```
ARMulif_SetSignal( &(amp;state->coredesc), RDIPropID_ARM_Signal_IRQ, TRUE );
```

This function asserts the emulated interrupt line (sets it to logic 0). It must be de-asserted after the processor has taken the IRQ exception.

The preferred method of clearing interrupt sources is for the interrupt handler to cause the interrupt source to be cleared. (See section 4 Example: Parallel Port Model).

3.2.2 FIQ

Provided that the CPSR F bit (bit 6) is 0, an FIQ may be generated by calling the function

```
ARMulif_SetSignal( &(amp;state->coredesc), RDIPropID_ARM_Signal_FIQ, TRUE );
```

This function asserts the emulated fast interrupt line (sets it to logic 0). It must be de-asserted after the processor has taken the IRQ exception.

The preferred method of clearing interrupt sources is for the interrupt handler to cause the interrupt source to be cleared. (See section 4 Example: Parallel Port Model).

3.2.3 Abort

To generate a data abort, the model must return `PERIP_DABORT`.

3.3 Event scheduling

When modeling peripherals, you may need the ARMulator to simulate the occurrence of external events which occur at a specific time or at some number of cycles into the future. ARMulator has the following routines to assist with the scheduling of such events:

```
ARMulif_Time
```

```
ARMulif_ScheduleNewTimedCallback
```

```
ARMulif_ScheduleTimedFunction
```

```
ARMulif_DescheduleTimedFunction
```

These are explained below. Note that as with all ARMulator functions, an `RDI_ModuleDesc` parameter is passed, allowing multiple instances of the peripheral, each with a different state.

3.3.1 ARMulif_Time

```
ARMTIME ARMulif_Time( RDI_ModuleDesc *mdesc )
```

This returns the number of clock ticks executed since system reset. You may wish to use this value for your own event scheduler.

3.3.2 ARMulif_ScheduleNewTimedCallback

```
void *ARMulif_ScheduleNewTimedCallback(
    RDI_ModuleDesc *mdesc, ARMul_TimedCallbackProc *func,
    void *handle, ARMTIME when, ARMTIME period);
```

This allows a function to be called a number of cycles in the future, where:

`func` is the function to be called

`when` is the cycle count at which the event should occur. This can be based upon the current cycle obtained using `ARMulif_Time`.

`period` This parameter is reserved for future use and should always be zero.

3.3.3 ARMulif_ScheduleTimedFunction

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

where:

`mdesc` is the handle for the core.

`tcb` is the handle for you to use if you want to deschedule the function.

3.3.4 ARMulif_DescheduleTimedFunction

```
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.

Note If you reset an interrupt line to 0 to cause an interrupt, then you must also set it to 1 again, otherwise interrupts will not stop. See 4 Example: Parallel Port Model.

You may use `ARMulif_ScheduleNewTimedCallbackOrARMulif_ScheduleTimedFunction` to call your own functions, provided that they match the prototype given in `simplelinks.h`:

```
typedef void (ARMul_TimedCallBackProc)(void *handle);
```

For example, you may schedule an event for `MyFunction` in the same way as for `DoAIRQ` or `DoAFIQ`. In the example below, the function `MyFunction` reschedules a call to itself in 500 cycles. It assumes that a state structure has previously been declared using the `BEGIN_STATE_DECL(MyModule)` which declares a structure `MyModuleState`.

```
extern void MyFunction( void *handle )
{
    ARMTime Now, delay, nextEventTime;

    MyModuleState state = (MyModuleState)handle;
    Hostif_ConsolePrint( state->hostif, "MyFunction\n" );

    Now = ARMulif_Time(&state->coredesc);
    delay = 500;
    nextEventTime = Now + delay;

    ARMulif_ScheduleNewTimedCallback( &state->coredesc, MyFunction,
    state, nextEventTime, 0 );
    /* Call this function again in 500 cycles by re-scheduling the
    event that calls it */
}
```

If more than one event is scheduled for a given time, the callbacks are stacked. When the specified clock cycle occurs then the callbacks are executed in reverse order of being called.

4 Example: Parallel Port Model

You can model the behavior of target hardware by making modifications to a copy of the ARMulator source code. This section describes how to emulate an example of a parallel port peripheral that causes an interrupt to occur and then places a character into a memory location from a text file (in effect, a model of data being received into a parallel port memory location).

The ARM debuggers do have a mechanism for generating IRQ or FIQ interrupts via the `$irq` and `$fiq` internal variables. An RDI target can export these variables to provide a means of asserting an interrupt request pin. See the *ADS 1.2 AXD and armsd Debuggers Guide* (ARM DUI 066D) section 5.5.8 for details on internal variables. If you are using RVD, the IRQ and FIQ internal variables are found on the Debug tab of the register pane. These variables are not accessible if you are using a LocalHost connection. See the *RVD User Guide* (ARM DUI 0153C) section 6.1.6.

You can also model external interrupts using one of the scheduling functions which calls another function a number of clock ticks in the future. (See section 3.5 Event Scheduling). This example uses the `ARMulif_ScheduleNewTimedCallback` function to call another function at 20,000 clock ticks in the future, which raises an IRQ exception. The function `ARMulif_ScheduleNewTimedCallback` is activated when the application program accesses a particular memory location.

The application program IRQ exception handler then reads in a character from another predefined location and clears the IRQ condition from the parallel port.

In the example, the parallel port clears the IRQ when a byte is read in from the port address. At this point, the new ARMulator model code clears the IRQ and schedules another interrupt to occur using the `ARMulif_ScheduleNewTimedCallback` function as before.

The application program installs an IRQ exception handler by setting the IRQ vector to be a branch instruction to the application IRQ handler. This is described in the *Debug Target Guide* or *RV ARMulator ISS User Guide*, section 4.5 Exceptions.

4.1 Creating the peripheral model (Windows procedure)⁴

- 1 Copy the file `parallel.c` from the supplied example code and save it in the ARMulator extension kit directory. This location is defined in section 1.2. You can download this and all of the other example code in this Application Note from the ARM Technical Documentation site at <http://www.arm.com/arm/documentation>.
- 2 Copy one of the `<MODEL_NAME>.b` directories and rename it to `parallel.b`.
- 3 Edit the `makefile` inside the `parallel.b` subdirectory called `intelrel` and replace all occurrences of `<MODEL_NAME>` with `'parallel'`.
- 4 Change your current directory to the location of your `makefile`.
- 5 Type the name of your `make` utility. On Windows, this is normally `nmake`.
- 6 On Windows, `parallel.dll` appears in:
`install_path\ARMulate\armulext\parallel.b\intelrel`
- 7 If you are using ADS, move `parallel.dll` to `install_path\bin` on Windows. This is where ARMulator expects to find models. If you are using RV ARMulator ISS, see section 1.1 for the location of the `bin` directory.

To complete this step, create a text file called `pport.txt`. It may contain whatever message you wish, but it must contain a full stop (.) character to terminate the loop. This

⁴ The procedure for building on Solaris, Linux or HP-UX is outlined in section 3.1.2 above.

file must be placed in the same directory as the `partest.axf` image (built below in section 4.4).

4.2 Explanation

The two function prototypes at the start of `parallel.c` define the following:

```
Parallel_Access
```

This method is the memory access callback that is executed whenever the ARMulator default memory model detects a memory access at the registered addresses (see the end of this section).

```
pport_set_irq
```

The function is scheduled to occur after reading from one of the registered memory addresses. Its purpose is to indicate to the debugger console that an IRQ has occurred and to reset the IRQ.

Between the `BEGIN_STATE_DECL(Parallel)` and `END_STATE_DECL(Parallel)` macros, two private data members are declared.

```
int pport_IRQ;           Determines whether or not the IRQ has been set.
```

```
FILE *pportfile;        A handle to a text file from which parallel port data is taken.
```

The macro defines a structure called `ParallelState` which includes the above attributes in addition to several others.

An instance of this structure is created by the `BEGIN_INIT(Parallel)` macro and a pointer `ParallelState *state` is assigned its address. This structure is passed to all callback functions relating to the model.

Before `END_INIT(Parallel)` is called, the member variables of `*state` are initialized and the peripheral is registered and assigned to an address range on the bus. This is performed using `ARMulif_ReadBusRange` to fill a structure of type `ARMul_BusPeripAccessRegistration`. During this call, the base address is read from the configuration file `peripherals.ami`. The memory access function is assigned to the `access_func` member and its capabilities are specified via the `capabilities` member. The peripheral is registered using `bus_registerPeripFunc` (a function pointer set up by `ARMulif_ReadBusRange`).

The memory access function is declared as:

```
static int Parallel_Access(void *handle, struct ARMul_AccessRequest *req)
```

It is called whenever a memory access falls within the range of the parallel port. `handle` points to the state structure and `req` points to a structure that provides memory access type, data and address. If the memory address is successfully read then `Parallel_Access` must return `PERIP_OK`. Otherwise, the address was not a part of the model and is not decoded so `PERIP_NODECODE` is returned.

A read from address `0x123450` opens the text file `pport.txt` and schedules an IRQ to occur in 20,000 cycles. An arbitrary data value is returned.

A read from address `0x123460` schedules another interrupt and sets the data word to the next character code from the text file.

The `#define` and `#include` directives at the end of the file are used to manage how the module is loaded when the debugger starts.

4.3 ARM application code

You must now use some sample ARM application code to process the characters arriving at a parallel port. This is called `partest.c` and is included with the Application Note 32 example code.

The application starts with a volatile global variable definition. A volatile qualified type indicates that something other than the application program can access or alter the value stored in the variable. The IRQ handler simply reads a character from memory location 0x123460 and places this in the global variable.

The `Install_Handler` code installs a branch instruction in the IRQ exception vector to branch to the IRQ handler.

The main function installs the IRQ handler then causes an access to memory location 0x123450 to initialize the interrupts. Immediately prior to reading this memory, a short inline assembly routine is called to enable interrupts in the ARM core CPSR (current program status register). This sets the IRQ disable flag in the CPSR to zero, in order to enable IRQ interrupts.

The program then goes into a loop until the first interrupt occurs, at which point program flow diverts to the IRQ handler. This updates the value of the global variable `globvar`. The value placed into this variable is then displayed on the screen.

The main program exits when it reads in the full stop termination character.

Note: When you single-step through the code, program flow does not appear to enter the IRQ handler code. To do this, you need to put a breakpoint on the IRQ function itself.

4.4 Running the application

The following steps must be performed before the application can be run.

- 1 Create a file called `parallel.dsc` and place it in `install_path\bin` on Windows, or `<install_path>/<platform>/bin`, where `<platform>` is one of `solaris`, `hpux` or `linux`. If you are using RVARMulator ISS, move the file to the location given in section 1.2. It must contain the following and be formatted as below:

```
;; ARMulator configuration file type 3
{ Peripherals
  {Parallel
    MODEL_DLL_FILENAME=Parallel
  }
  {
    No_Parallel=Nothing
  }
}
```

If you are using ADS 1.1, add a `META_SORDI_DLL=Parallel` entry after the `MODEL_DLL_FILENAME=Parallel` entry.

- 2 Load the `default.ami` file into a text editor and add a reference to your model:

```
{Parallel=Default_Parallel
}
```

- 3 Load the `peripherals.ami` file into a text editor and add a configuration section for your model.

```
{ Default_Parallel=Parallel
Range:Base=0x123450
}
```

The base address and size of the peripheral in the memory map is determined by the `ARMulif_ReadBusRange` call in `parallel.c`. In this case the base is set to `0x1234560` and the size is `0x20`.

- 4 Compile the file `partest.c` using the `armcc` compiler, as follows:

```
armcc -g -opartest.axf partest.c
```

This can be loaded into `armsd`, `ADW` or `AXD` and the example executed.

Note You cannot use a cached core target for this example unless you first disable `pagetables`.

4.5 Running the Example on RealView Debugger

If you are using RealView Debugger, you must invoke it from the same directory in which the `pport.txt` file exists. If this is not done, RVD will not be able to open the file and will report an error.

On Windows, open an Explorer window and locate the RVD bin directory (normally somewhere similar to `C:\Program Files\ARM\RVD\Core\1.6.1\159\win_32-pentium\bin\`). Right click on `rvdebug.exe` and select "Create shortcut" from the resulting context menu. Drag the shortcut to the desktop. Right click on the icon and select "Properties" from the context menu, then choose the "Shortcut" tab in the resulting dialog. In the "Start in" field, enter the path of the ARMulator bin directory. In the case of RV ARMulator ISS, this is similar to

`C:\ProgramFiles\ARM\RVARMulator\ARMulator\1.3.1\1310.20030312\win_32-pentium`. In the case of ADS, this is similar to `C:\Program Files\ADSV1_2\bin`

Even if you have built the example for Solaris or Linux, you cannot run this example on these platforms through RVD because there is no way to specify the location from which RVD is started.

5 Example: Exception Generator Memory Model

This example of a memory model provides the ability to generate interrupts immediately and to schedule them for a later time.

5.1 Creating and modifying the files

Follow the procedure set out in section 4.1 (Adding a memory or peripheral model) substituting “projectx” for “parallel”.

5.2 Writing code to access the memory locations

Since the peripheral model is being employed in this example as per the previous one, it is possible to use the same program structure. The changes involved are:

- Remove the additional member variables `pport_IRQ` and `pportfile` from the `BEGIN_INIT ... END_INIT` block
- Change the memory access function name (and its prototype) to `ProjectX_Access`
- Replace the code in the `ProjectX_Access` function.

There are four separate memory trigger locations:

0x200000	Writing 1 here causes an IRQ. Writing 2 here causes an FIQ.
0x200004	Writing a value here schedules an IRQ in value cycles.
0x200008	Writing a value here schedules an FIQ in value cycles.
0x20000C	Writing 1 here clears the IRQ. Writing 2 here clears the FIQ.

5.3 Writing ARM application code

You need some application code to exercise this memory model. This code is provided in the example file `interrupts.c`.

Build an ARM executable image using . the following command:

```
armcc -g -o interrupts.axf interrupts.c
```

5.4 Running the application

Please refer to section 4.4, replacing “parallel” with “projectx” where appropriate. The entry required for `peripherals.ami` is:

```
{ Default_ProjectX=ProjectX
; ; as per AppNote 32
Range:Base=0x200000
}
```

Running the code in a debugger should give similar results to the following:

```

Installing handlers
Handlers installed
Cause an irq
IRQ requested
Write to 0x200000 - value = 00000001
IRQ cleared
Write to 0x20000C - value = 00000001
Cause an FIQ
FIQ requested
Write to 0x200000 - value = 00000002
FIQ cleared
Write to 0x20000C - value = 00000002
Cause an irq
IRQ requested
Write to 0x200000 - value = 00000001
IRQ cleared
Write to 0x20000C - value = 00000001
IRQ scheduled in 600 cycles
Write to 0x200004 - value = 00000258
1
IRQ cleared
Write to 0x20000C - value = 00000001
2
3
4
5
6
7
8
9
FIQ scheduled in 1500 cycles
Write to 0x200008 - value = 000005dc
1
2
3
4
FIQ cleared
Write to 0x20000C - value = 00000002
5
6
7
8
9
IRQ scheduled in 1000 cycles
Write to 0x200004 - value = 000003e8
FIQ scheduled in 1000 cycles
Write to 0x200008 - value = 000003e8
1
2
FIQ cleared
Write to 0x20000C - value = 00000002
IRQ cleared
Write to 0x20000C - value = 00000001
3
4
5
6
7
8
9

```

6 Example: Coprocessor Model

This example of a coprocessor model provides the ability to generate interrupts and to schedule them for a later time. This is not a typical hardware application and simply illustrates how to structure a coprocessor model.

6.1 Creating the files

Source code for `mycopro.c` is provided with the Application Note 32 example code.

6.2 Editing files

Follow the procedure set out in section 4.1 (Adding a memory or peripheral model) substituting “mycopro” for “parallel”. Then refer to section 4.4, replacing “parallel” with “mycopro” where appropriate. The entry required for `peripherals.ami` is:

```
{ Default_Mycopro=Mycopro
}
```

6.3 Writing application files

You need some application code to exercise this peripheral model. This code is provided in two parts. The first part is written in assembler and is the file `copro.s`. This handles coprocessor data operations. The second part is written in C and is the file `coprotest.c`. This demonstrates calling from C the functions defined in assembler in `copro.s`.

Build the two files `copro.s` and `coprotest.c` using the following commands:

```
armasm -g copro.s -o copro.o
armcc -g -c coprotest.c -o coprotest.o
armlink copro.o coprotest.o -o coprocess.axf
```

6.4 Running the code

Running the code should give similar results to the following:

```
Installing handlers
Handlers installed
Use MRC/MCR
Value = ffffffff
Value = aaaaaaaaaa
CDP 1 - cause a FIQ
ExpFIQ entered - clear source
ecting a fiq.
CDP1 done

CDP 2 - cause an IRQ
ExpIRQ entered
ecting an irq.
CDP2 done

Read timer
Value = 00008ab6
Value = 00009504
```

7 Debugging ARMulator models in Visual C++

The debugging facilities of Visual C++ may be employed for faultfinding in ARMulator models. The execution of customized dynamic-link libraries can be examined at all stages from initialization through to memory/peripheral or coprocessor register callbacks.

The following instructions outline the procedure necessary to debug an ARMulator model in Visual C++. These instructions apply to Visual C++ versions 5 and 6. You must compile a build which links against the correct C runtime library to avoid memory conflicts.

The required stages involved in model debugging are:

1. Create a Visual C++ dynamic-link library project.
2. Add files to the project.
3. Configure compiler settings, library and header file locations.
4. Compile the module.
5. Ensure `.dsc` and `.ami` configuration files have been properly configured.
6. Set breakpoints.
7. Launch a debugger via Visual C++.

7.1 Creating a project

It is assumed that you have a directory structure organized in the same way as the examples in this Application Note. This means that a subdirectory named `MyModel.b` exists containing your model as well as a corresponding makefile. To create a `.dll` project:

- Launch Visual C++.
- Choose File->New... from the menu bar.
- Choose Win32 Dynamic-link library as the project type and enter your module name in the 'Project Name' box. Specify location as the folder `MyModel.b`.
- When prompted for the kind of DLL to create, choose "An empty DLL project".
- Hit OK when prompted.

7.2 Adding files

Add the following files to the project.

- Your model source code (`MyModel.c`).
- The files `sordi.def` and `version.rc` from the `armulext` subdirectory.

7.3 Configure project settings

First you need to create a DebugRelease build type.

- From the Build menu, select Configurations -> New. Click on the Add button.
- In the resulting dialog, type "DebugRelease" in the Configuration box. "Copy settings from" the Debug project settings. Close the configuration box.
- From the Build menu, now set the Active Configuration to DebugRelease.

Now configure the project settings.

- Open the settings dialog by choosing the menu Project->Settings... or pressing ALT+F7.
- Edit the "Settings For" dropdown field to "DebugRelease".
- On the C/C++ tab, in the dropdown Category Preprocessor, check that the following are entered in "Additional include directories": `..\..\rdi`, `..\..\clx`, `..\..\armulif`
- On the C/C++ tab, in the dropdown Category Code Generation, select "Multithreaded DLL" as the Run-time library. This sets the correct C library for release builds.
- On the Link tab, in the dropdown Category General, check that the Object/library entry has been replaced with: `..\..\clx\clx.b\intelrel\clx.lib`
`..\..\armulif\armulif.b\intelrel\armulif.lib`
- On the Link tab, in the 'output file name' box, enter `install_path\bin\mymodel.dll`.
- On the Debug tab, configure the executable to point to the armsd, AXD or RVD executable.

7.4 Compile the module

Choose menu option Build->Build mymodel.dll or press F7.

7.5 Ensure .dsc and .ami configuration files have been properly configured

Follow the procedure set out in section 3.1.2 to ensure that the correct entries have been made in `default.ami` and `peripherals.ami`. Your model's `.dsc` file should reside in the ARMulator executables directory (see section 3.1.2 step 7 for ADS, or section 1.2 for RVARMulator ISS).

7.6 Set breakpoints

Open your model's source code file and set breakpoints on lines where you wish to examine access to the module.

7.7 Launch debugger

Pressing F5 will launch the debugger you chose in step 3. By loading an image into the debugger and executing it, any model functions accessed which contain breakpoints will halt execution and the VC++ debugger may be used to examine model state.

8 Calling a Peripheral Every Cycle

An increasingly common technique when designing a peripheral is to perform some operations on every single memory cycle. This is not the recommended method of designing a model due to inherent inefficiencies. It is often simpler and less processor intensive to use the scheduling functions (see 3.3 Event Scheduling) to achieve the same result. For an example of this, study the timer peripheral (`timer.c`) provided with ADS.

The following example peripheral illustrates how to obtain a callback every memory cycle. Using the same procedures as described for the previous example models, enter the following C source then build the peripheral model. This model is based upon `Tracer.c` which is provided with ADS.

`Cycles.c`:

```
/*
** Copyright (C) ARM Limited, 2003. All rights reserved.
*/

/* everycycle.c - function TraceX called every cycle
*/

#include "minperip.h"
#include "armul_mem.h"
#include "armul_callbackid.h"

#if !defined(NDEBUG)
# if 1
# else
#  define VERBOSE
# endif
#endif

BEGIN_STATE_DECL(cycles)
    ARMul_MemInterface child, *mem_ref, bus_mem;
END_STATE_DECL(cycles)

static int TraceBusMemAccess(void *handle,
                             ARMword address,
                             ARMword *data,
                             ARMul_acc access_type);

static int TraceX(cyclesState *ts, ARMword addr, uint32 *data, int rv,
                  unsigned acc);

static unsigned TraceMemInfo(void *handle, unsigned type, ARMword *pID,
                             uint64 *data);
static ARMTime TraceReadClock(void *handle);
static const ARMul_Cycles *TraceReadCycles(void *handle);
static uint32 TraceGetCycleLength(void *handle);
static int RDI_info(void *handle, unsigned type, ARMword *arg1, ARMword *arg2);

BEGIN_INIT(cycles)
    Hostif_PrettyPrint(state->hostif, config, " , everycycle");
    {
        /* Now register the access function */
        ARMul_MemInterface *mif;
        uint32 ID[2];
        ID[0] = ARMulBusID_Core;
        ID[1] = 0;
        mif = ARMulif_QueryMemInterface(&state->coredesc, &ID[0]);

        assert( mif );
    }
}
```



```

if( mif ) {
    state->bus_mem.handle = state;
    state->bus_mem.x.basic.access = TraceBusMemAccess;

    state->bus_mem.mem_info=TraceMemInfo;
    state->bus_mem.read_clock=TraceReadClock;
    state->bus_mem.read_cycles=TraceReadCycles;
    state->bus_mem.get_cycle_length = TraceGetCycleLength;

    /* </> */

    switch(mif->memtype)
    {
    case ARMul_MemType_Basic:
    case ARMul_MemType_16Bit:
    case ARMul_MemType_Thumb:
    case ARMul_MemType_BasicCached:
    case ARMul_MemType_16BitCached:
    case ARMul_MemType_ThumbCached:
    case ARMul_MemType_ARMissAHB:
        break;

    case ARMul_MemType_StrongARM:
        /* (state->bus_mem.x.strongarm.core_exception = TraceCoreException;
        state->bus_mem.x.strongarm.data_cache_busy = TraceDataCacheBusy; */

        state->bus_mem.x.strongarm.core_exception = NULL;
        state->bus_mem.x.strongarm.data_cache_busy = NULL;
        break;
    case ARMul_MemType_ARM8:
/*
        state->bus_mem.x.arm8.core_exception = TraceCoreException;
        state->bus_mem.x.arm8.access2 = TraceBusMemAccess2;

state->bus_mem.x.arm8.core_exception = NULL;
state->bus_mem.x.arm8.access2 = NULL;
        break;

    case ARMul_MemType_ARMissCache:
    case ARMul_MemType_ARM9:
    case ARMul_MemType_ByteLanes:
    default:
        break;
    }
}

ARMulif_InstallUnkRDIInfoHandler(&state->coredesc,
                                RDI_info,state);

ARMul_InsertMemInterface(mif,
                        &state->child,
                        &state->mem_ref,
                        &state->bus_mem);
}
END_INIT(cycles)

BEGIN_EXIT(cycles)
END_EXIT(cycles)

static int TraceBusMemAccess(void *handle,
                            ARMword address,
                            ARMword *data,
                            ARMul_acc access_type)
{
    cyclesState *ts = (cyclesState *)handle;
    int err =
        ts->child.x.basic.access(ts->child.handle,address,data,access_type);
}

```

```

        TraceX(ts, address, data, err, access_type);
        return err;
    }

    static int TraceX(cyclesState *ts, ARMword addr, uint32 *data, int rv,
                      unsigned acc)
    {
        /* DEBUG catch ALL accesses
         * if ((addr<ts->range_lo || (addr>=ts->range_hi && ts->range_hi!=0)))
         * {
         * }
         */

        /* display diagnostic message */

        static ARMTIME prevtime = 0;
        static ARMTIME currtime = 0;

        currtime = ARMulif_Time(&ts->coredesc);

        if( currtime != prevtime ) {
            /* INSERT YOUR HANDLER HERE */
            prevtime = currtime;
            Hostif_ConsolePrint(ts->hostif,"Cycle: %u ", currtime );
            Hostif_ConsolePrint(ts->hostif,"Access address: %u\n", addr );
        }

        return rv;
    }

    /* ----- */

    static int RDI_info(void *handle,unsigned type,ARMword *arg1,ARMword *arg2)
    {
        return RDIError_UnimplementedMessage;
    }

    static unsigned TraceMemInfo(void *handle, unsigned type, ARMword *pID,
                                  uint64 *data)
    {
        cyclesState *mem = (cyclesState *)handle;
        if (mem->child.mem_info)
        {
            return mem->child.mem_info(mem->child.handle,type,pID,data);
        }
        else
        {
            return RDIError_UnimplementedMessage;
        }
    }

    /* Aims to return a value in microseconds */
    static ARMTIME TraceReadClock(void *handle)
    {
        cyclesState *mem = (cyclesState *)handle;
        if (mem->child.read_clock)
        {
            return mem->child.read_clock(mem->child.handle);
        }
        else
        {
            return 0L;
        }
    }

    static const ARMul_Cycles *TraceReadCycles(void *handle)

```

```

{
    cyclesState *mem = (cyclesState *)handle;
    if (mem->child.read_cycles)
    {
        return mem->child.read_cycles(mem->child.handle);
    }
    else
    {
        return NULL;
    }
}

static uint32 TraceGetCycleLength(void *handle)
{
    cyclesState *mem = (cyclesState *)handle;
    if (mem->child.get_cycle_length)
        return mem->child.get_cycle_length(mem->child.handle);
    /* Todo: Otherwise guess from CCFG and CPUSPEED */
    return 0;
}

/*--- <SORDI STUFF> ---*/

#define SORDI_DLL_NAME_STRING "cycles"
#define SORDI_DLL_DESCRIPTION_STRING "cycles (test only)"
#define SORDI_RDI_PROCVEC cycles_AgentRDI
#include "perip_sordi.h"

#include "perip_rdi_agent.h"
    IMPLEMENT_AGENT_PROCS_NOEXE_NOMODULE(cycles)
    IMPLEMENT_AGENT_PROCVEC_NOEXE(cycles)

/*--- </> ---*/

/* EOF cycles.c */

```

Your code to be called every cycle should be placed at the comment:

```
/* INSERT YOUR HANDLER HERE */
```

Whenever the cycle count is incremented whilst you are running a program in the debugger, the cycle number and memory address accessed is output to the console window.

9 Communication between ARMulator models

When developing a software simulation of your embedded system, you may have two or more ARMulator models that need to communicate with each other. An important issue is how to provide a common memory view for the models

AXD can only support a single instance of ARMulator per session. This means that multiple ARMulators can only be run in isolation under separate AXD processes. This would be the case if you were trying to model a multiple core system, for example. In RVD, you can connect to more than one ARM simulator target simultaneously from the same debugger if you are simulating a multi-core environment. This functionality is available in RVD 1.6.1 using the RealView ARMulator Instruction Set Simulator, provided that you have a multiple core license. If you were instead modelling a single core but with a number of peripherals that are intended to have shared memory space, you still need to ensure that the memory view of each model is consistent. Shared memory must be implemented as described below.

9.1 How to model shared memory

If you wish to have a shared memory region between multiple ARMulators or ARMulator peripherals, then you need to implement an ARMulator extension model for each ARMulator instance. The procedure for adding such models is described above in section 3.

The Windows API has a concept of a "shared memory-mapped file" which can be opened by a number of different processes in a system. Such a file can be opened in each ARMulator to model the memory shared between the processors. This technique is implemented by the "LCD viewer" application (Application Note 92, available from www.arm.com) to access the LCD bitmap from the ARMulator LCD model. The LCD model can be used as a basis to extend for sharing memory between multiple ARMulators or peripherals.

The debugger refreshes memory windows when execution stops (e.g., breakpoint hit or single step complete). The debugger requests the memory values using a special "non-accounted" memory access type which avoids the ARMulator cycle counts being updated. The memory will not be updated in a debugger's window until the user does a single-step/go or opens a new memory window in the debugger.

There is no direct support for modelling of a shared memory map between two ARMulators or peripherals, as each will execute as an individual process on the host OS, but in theory there are a number of possibilities:

- (i) Adjust the application (if needed) so that you can easily encapsulate all accesses to this shared memory location. This could then be intercepted and replaced with alternative functionality to read and write from a shared file managed by the host OS.
- (ii) Use the debug tools to place break/watchpoints on memory accesses in the application, the breakpoint could then invoke a macro to update its own copy of the memory, dump the memory to a file, then cross-trigger to the other debugger so that it can update its own memory from the shared file.
- (iii) Model the shared memory location as an ARMulator peripheral. This requires more work, but is a better solution. Here any accesses to the shared memory will be intercepted and handled in whichever way you wish, typically using a OS specific features such as Win32 shared files to maintain the status of the shared memory.
- (iv) Model a system process controller. This involves the most work of all. Here every single operation of each ARMulator is monitored by the process controller, accesses to the shared memory location can again be handled as you wish. This method would ensure extremely tight cohesion between the two ARMulator processes.

Solutions (i) and (ii) offer crude and simplistic approaches and are probably easier to implement but would typically provide less accurate (if acceptable) results. Solutions (iii) and (iv) offer more complete approaches to the problem, but will require more work. The accuracy of (iii) is highly dependent upon the application being modelled.

9.2 How to coordinate execution

AXD can only support a single instance of ARMulator per session. This means that multiple ARMulators can only be run in isolation unless you implement a shared memory map file as described above.

In contrast, RVD supports cross-triggering via software as standard. When an event, such as a breakpoint being hit, occurs on one target, a message is sent to the other target connection to send the core into debug state. Further information on this feature is documented in the *RVD Extensions User Guide* (DUI 0174D-08), Chapter 5.

10 Known Issues

If on starting AXD you are presented with “RDI Warning 00129: Can’t initialize” then it is likely that you have an incorrect ARMulator configuration file. Check your model `.ami` and `.dsc` files and restart the debugger. See the section relating to the model for the required changes to `peripherals.ami` and `default.ami`, and use the supplied `.dsc` files for reference.

If on connecting to ARMulator via localhost in RVD, you get an error which says, “Error V20013 (Vehicle): Unable to connect to emulator/simulator. See Output log for details” followed by, “Failed on remote SIM/EMU/EVM: OpenAgent failed (UnableToInitialise) with a non-null agent handle!”, there is probably an error in your model `.ami` and `.dsc` files. If after dismissing the error dialogs you find an error on the Cmd tab which states, “A child of Flatmem failed to initialize, error 129.” then there is likely a problem in the model’s `.dsc` file. If instead the Cmd tab only reports a very short ARMulator header, then the fault is probably in the `peripherals.ami` or `default.ami` files. In any case, close RVD, ensure that RVBroker has been shut down, correct the error, then restart RVD.