Application Note 32

The ARMulator

Document number: ARM DAI 0032E
Issued: August 2001

ARM

Application Note 32
Title

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
Aug 1996	A	First release
Aug 1996	B	Changes to remove references to integration with hardware modeling environments.
Jan 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.

Proprietary notice

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 errata@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 addtions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

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 accurate benchmarking of ARM-targeted software (though its performance is somewhat slow compared to real hardware)
- 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 the ARM debuggers to provide a hardware-independent ARM software development environment. Communication takes place via the Remote Debug Interface (RDI).

For full details, refer to:

- Debug Target Guide (ARM DUI 0058C)

Note This Application Note is designed for ARM Developer Suite 1.1. For information on extending the ARMulator for SDT or ADS 1.0 please see the previous revisions of AppsNote 32.

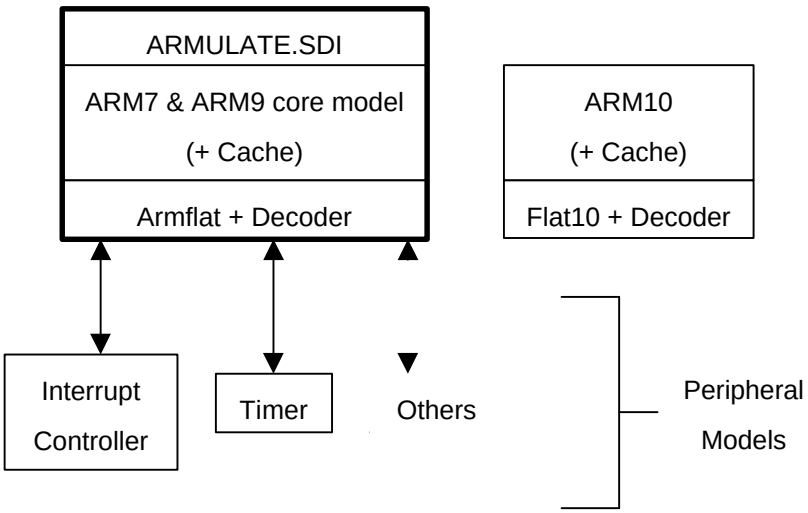
This document will apply to ADS 1.2 but the documentation reference numbers are likely to change. Please see Appendix A for known changes required for version 1.2.

2 The Structure of the ARMulator

The ARMulator comprises several parts:

- A model of the ARM processor core and cache (if used)
- A base memory model (armflat) incorporating address decoding. This causes the relevant peripheral model to be accessed when memory within its registered range is addressed.
- 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[1].



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[2]) and the peripheral decoder. The decoder and flat memory model are integral to the ARMulator and cannot be modified. However, this is not the case in ADS 1.2. Please see section Appendix A for details.

[1] Note that this structure has changed in ADS 1.2. See Appendix A for details.

[2] There is a known bug in ADS 1.1 whereby addresses lying between 2GB and 4GB may not be read from a configuration file. This has been fixed in ADS 1.2.

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* (ARM DUI 0058C) ***sections 2.6 and 2.7*** for more information.

You may add additional functions between the `BEGIN_INIT … 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* (ARM DUI 0058C) ***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* (ARM DUI 0058C), ***section 4.13, "Map files"***.

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 supplied coprocessor model (dcc.c) provides a model of a debug communications coprocessor. This model is fully customizable and models of other coprocessors can be added easily. The process for adding a coprocessor model is described ***in 3.1.3 Adding a coprocessor model***.

2.4 The operating system interface

The operating system model is called whenever the ARM executes a 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* ARM DUI 0058C).

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* (ARM DUI 0058C) for a full description of the Application Programming Interface (API) between the ARM debugger and the memory model, coprocessor model and operating system.

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 (SWI) handlers to respond from SWIs generated by your application.

Refer to the *Debug Target Guide* (ARM DUI 0058C), ***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* (ARM DUI 0058C), ***3.1.1 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* (ARM DUI 0058C), ***3.5 Configuring ARMulator to disable a model***.

3.1.2 Adding a memory or peripheral model

Unlike previous versions, the ARMulator supplied with ADS 1.1 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

1. If you have not already done so, make a working copy of the rebuild kit. Make your changes to the working copy, *not* the original files.
2. Place new sources for your memory or peripheral model in the source directory (`install_path\ARMulate\armulext`).
3. Create a new copy of one of the directories named `<MODEL_NAME>.b` in the source directory and rename it to reflect your model name (assumed to be `MyModel`).
4. Edit the Makefile inside the new directory's `IntelRel` subdirectory, replacing all occurrences of `<MODEL_NAME>` with your new model name, `MyModel`.
5. Change your current directory to `install_path\ARMulate\armulext\MyModel.b\intelrel`
6. Depending on your system: For Windows, type: `nmake` For UNIX, type: `make`
7. On Windows, mymodel.dll appears in: `install_path\ARMulate\armulext\MyModel.b\intelrel`. Move `mymodel.dll` to: `install_path\bin`. 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`.

Create a file called `MyModel.dsc` and place it in `install_path\bin`. It must contain the following:

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

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

```
{Tracer=Default_Tracer
}
```

Add the reference to your model:

```
{Tracer=Default_Tracer
}

{MyModel=Default_MyModel
}
```

Save your edited `default.ami` file.

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* (ARM DUI 0058C), ***4.15.2 File format*** for details of how to construct configuration files.

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( &(state->coredesc), RDIPropID_ARMSignal_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( &(state->coredesc), RDIPropID_ARMSignal_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_ScheduleNewTimedCallback or `ARMulif_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 provided from the rebuild area of the ARM Developer suite. This section describes how to emulate an example of a parallel port peripheral that causes an interrupt to occur and them 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. A target can export these variables to provide a means of asserting an interrupt request pin. See the *Debuggers Guide* (ARM DUI 066C), page 5-67 for details.

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 ***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* (ARM DUI 0058C), ***4.5 Exceptions***.

4.1 Creating the peripheral model

1. Create a new, empty file called parallel.c and save it in the directory install_directory\ARMulate\armulext.
2. Enter the following code in the file parallel.c (a full explanation follows):

```
/* Parallel.c - an example ARMulator peripheral model
 */

#include "minperip.h"
#include "armul_mem.h"
#include <stdio.h>

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

extern unsigned pport_set_irq(void* handle);

BEGIN_STATE_DECL(Parallel)
  int pport_IRQ;
  FILE *pportfile;

  /* store details of peripheral registration */
  ARMul_BusPeripAccessRegistration my_bpar;
END_STATE_DECL(Parallel)
```

```
BEGIN_INIT(Parallel)
Hostif_PrettyPrint(state->hostif, config, ", Parallel");
{
  /* Note that BEGIN_INIT macro defines ParallelState *state     */

  unsigned err;

  /* initialise state member variables */
  state->pport_IRQ=0;
  state->pportfile=NULL;

        err = RDIError_NoError;

    /* Provide memory access callback */
    state->my_bpar.access_func = Parallel_Access;
    state->my_bpar.access_handle = state;
    state->my_bpar.capabilities = PeripAccessCapability_Minimum;

    err = ARMulif_ReadBusRange(&state->coredesc, state->hostif,
                        ToolConf_FlatChild(config, (tag_t)"RANGE"),
                        &state->my_bpar,
                        0x123450,0x20,"");

    err = state->my_bpar.bus->bus_registerPeripFunc(BusRegAct_Insert,
                                                    &state->my_bpar);
    if (err)
        return err;
}
END_INIT(Parallel)


BEGIN_EXIT(Parallel)
END_EXIT(Parallel)

/* Memory access function */
static int Parallel_Access(void *handle, struct ARMul_AccessRequest *req)
{
    ARMWord address = req->req_address[0];
    ARMWord *data = req->req_data;
    unsigned type = req->req_access_type;
    ParallelState *state=(ParallelState *)handle;
    ARMTime Now, delay, nextEventTime;

    assert(address >= state->my_bpar.range[0].lo && address <= state-
>my_bpar.range[0].hi);

    /* We have identified a parallel port access */
    if( (address == 0x123450) && acc_READ(type) ) {
        Hostif_ConsolePrint( state->hostif, "Trying to open pport.txt.\n" );
        state->pportfile = fopen( "pport.txt", "rb" );
        if( state->pportfile == NULL ) {
              Hostif_ConsolePrint( state->hostif, "Error: Could not open pport.txt\n"
);
              Hostif_ConsolePrint( state->hostif, "Error: Could not open pport.txt\n"
);
        } else {
              Hostif_ConsolePrint( state->hostif, "pport.txt successfully opened.\n" );
              Hostif_ConsolePrint( state->hostif, "An interrupt has been scheduled.\n"
);

              Now = ARMulif_Time(&state->coredesc);
                    delay = 20000;
```

```
                        nextEventTime = Now + delay;

                        ARMulif_ScheduleNewTimedCallback(
                        &state->coredesc, pport_set_irq, state, nextEventTime, 0 );
            }

                  *data = 1234;
                  return PERIP_OK;
        }

        if( (address == 0x123460) && acc_READ(type) ) {
            Hostif_ConsolePrint( state->hostif, "Read from 0x123460\n" );
            if( state->pport_IRQ != 0 ) {
                  ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ, FALSE );
                  state->pport_IRQ = 0;
            }
            /* schedule another interrupt */
            Now = ARMulif_Time(&state->coredesc);
                  delay = 6000;
                  nextEventTime = Now + delay;

            ARMulif_ScheduleNewTimedCallback(
                        &state->coredesc, pport_set_irq, state, nextEventTime, 0 );

            if( state->pportfile )
                  *data = fgetc(state->pportfile);
            else
                  printf( "pportfile is null\n" );

            return PERIP_OK;
      }

      return PERIP_NODECODE;
    }

    extern unsigned pport_set_irq(void* handle)
    {
      /* state is obtained directly from ParallelState */
      ParallelState *state=(ParallelState *)handle;
      Hostif_ConsolePrint( state->hostif, "An IRQ has occured\n" );

      /* Assert the IRQ */
      ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ, TRUE );
      state->pport_IRQ = 1;
      return 1;
    }


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

    #define SORDI_DLL_NAME_STRING "Parallel"
    #define SORDI_DLL_DESCRIPTION_STRING "Parallel port model"
    #define SORDI_RDI_PROCVEC Parallel_AgentRDI
    #include "perip_sordi.h"

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

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

3 Copy one of the <MODEL_NAME>.b directories and rename it to parallel.b.

4 Edit the makefile inside the parallel.b subdirectory called intelrel and replace all occurances of <MODEL_NAME> with ‘parallel’.

5 Change your current directory to install_path\ARMulate\armulext\MyModel.b\intelrel

6 Depending on your system: For Windows, type: nmake For UNIX, type: make

7 On Windows, mymodel.dll appears in: install_path\ARMulate\armulext\MyModel.b\intelrel. Move mymodel.dll to: install_path\bin. This is where ARMulator expects to find models.

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 file must also be placed in the above directory.

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 later).

`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 Writing ARM application code

You must now write some sample ARM application code to process the characters arriving at a parallel port.

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.

Create a new text file called partest.c and put the following code in it:

```
#include <stdio.h>

__inline void enable_IRQ(void);

volatile int globvar;
void __irq myIRQhandler(void)
{
  char *portlocn = (char *)0x123460;
  globvar = *portlocn;
}

unsigned Install_Handler( unsigned routine, unsigned *vector )
{
  unsigned vec, oldvec;
  vec = ((routine - (unsigned)vector - 0x8) >> 2 );
  vec = 0xea000000 | vec;
  oldvec = *vector;
  *vector = vec;
  return (oldvec);
}

int main( void )
{
  unsigned *irqvec = (unsigned *)0x18;
  int *loc = (int*)0x123450;

  Install_Handler( (unsigned)myIRQhandler, irqvec );
```

```
  /* ENABLE IRQs – These are disabled by default in AXD under ADS 1.1 */
  enable_IRQ();

  printf( "Contents of location 0x123450 = %d\n", *loc );

  do {
        globvar = -1;

        while( globvar == -1 );

        printf( "Character read in from text file %c\n", globvar );
  }
  while( globvar != '.' );
  return 0;
}

__inline void enable_IRQ(void)
{
  int tmp;
  __asm
  {
    MRS tmp, CPSR
    BIC tmp, tmp, #0x80
    MSR CPSR_c, tmp
  }
}
```

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.

Create a file called `parallel.dsc` and place it in `install_path\bin`. It must contain the following:

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

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

```
{Parallel=Default_Parallel
}
```

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

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

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

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

This can be loaded into the debugger of your choice (armsd, ADW or AXD).

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.

The complete source listing for `projectx.c` is as follows:

```
/* ProjectX.c - Exception Generator Memory Model
 */

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

static int ProjectX_Access(void *handle,
                       struct ARMul_AccessRequest *req);

static void clearIrq( void *handle );
static void clearFiq( void *handle );
static void setIrq( void *handle );
static void setFiq( void *handle );

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

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

BEGIN_STATE_DECL(ProjectX)

  /* store details of peripheral registration */
  ARMul_BusPeripAccessRegistration my_bpar;

END_STATE_DECL(ProjectX)

BEGIN_INIT(ProjectX)
Hostif_PrettyPrint(state->hostif, config, ", ProjectX");
{
  /* Note that BEGIN_INIT macro defines ProjectXState *state     */

  unsigned err;

        err = RDIError_NoError;

    /* Provide access-callback */
    state->my_bpar.access_func = ProjectX_Access;
    state->my_bpar.access_handle = state;
    state->my_bpar.capabilities = PeripAccessCapability_Minimum;

    err = ARMulif_ReadBusRange(&state->coredesc, state->hostif,
                         ToolConf_FlatChild(config, (tag_t)"RANGE"),
                         &state->my_bpar,
                         0x200000,0x0D,"");

    err = state->my_bpar.bus->bus_registerPeripFunc(BusRegAct_Insert,
                                                    &state->my_bpar);
    if (err)
        return err;
}
END_INIT(ProjectX)


BEGIN_EXIT(ProjectX)
END_EXIT(ProjectX)

/* MemAccess functions */
static int ProjectX_Access(void *handle,
                        struct ARMul_AccessRequest *req)
{
    ARMWord address = req->req_address[0];
    ARMWord *data = req->req_data;
    unsigned type = req->req_access_type;
    ProjectXState *state=(ProjectXState *)handle;
    ARMTime Now, delay, nextEventTime;

    assert(address >= state->my_bpar.range[0].lo && address <= state-
>my_bpar.range[0].hi);

    if( (address == 0x200000) && acc_WRITE(type) ) {
        switch(*data) {
               case 1:
                      Hostif_ConsolePrint( state->hostif, "IRQ requested\n" );
```

```
                        ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ,
TRUE );
                        break;

                case 2:
                        Hostif_ConsolePrint( state->hostif, "FIQ requested\n" );
                        ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_FIQ,
TRUE );
                        break;
        }

        Hostif_ConsolePrint( state->hostif, "Write to 0x200000 - value = %08x\n", *data
);
        return PERIP_OK;
  }

  /* IRQ scheduling */
  if( (address == 0x200004) && acc_WRITE(type) ) {
        delay = *data;
        Hostif_ConsolePrint( state->hostif, "IRQ scheduled in %d cycles\n", delay );

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

        ARMulif_ScheduleNewTimedCallback(
                        &state->coredesc, setIrq, state, nextEventTime, 0 );

                /* DEBUG schedule event for the same time to see which happens
                ARMulif_ScheduleNewTimedCallback(
                        &state->coredesc, dummyCallback, state, nextEventTime, 0 );

                 */

                Hostif_ConsolePrint( state->hostif, "Write to 0x200004 - value = %08x\n",
delay );

        return PERIP_OK;
  }

  /* FIQ scheduling */
  if( (address == 0x200008) && acc_WRITE(type) ) {
        delay = *data;
        Hostif_ConsolePrint( state->hostif, "FIQ scheduled in %d cycles\n", delay );

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

        ARMulif_ScheduleNewTimedCallback(
                        &state->coredesc, setFiq, state, nextEventTime, 0 );

                Hostif_ConsolePrint( state->hostif, "Write to 0x200008 - value = %08x\n",
delay );

        return PERIP_OK;
  }

  /* Interrupt Clearing */
  if( (address == 0x20000C) && acc_WRITE(type) ) {
        switch(*data) {
                case 1:
                        Hostif_ConsolePrint( state->hostif, "IRQ cleared\n" );
                        clearIrq(state);
                        break;
```

```
                case 2:
                        Hostif_ConsolePrint( state->hostif, "FIQ cleared\n" );
                        clearFiq(state);
                        break;
        }

        Hostif_ConsolePrint( state->hostif, "Write to 0x20000C - value = %08x\n", *data
);
        return 1;
  }

  /* did not decode the address */
  return PERIP_NODECODE;
}

static void clearIrq( void *handle )
{
  ProjectXState *state = (ProjectXState *) handle;
  ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ, FALSE );
}

static void clearFiq( void *handle )
{
  ProjectXState *state = (ProjectXState *) handle;
  ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_FIQ, FALSE );
}

static void setIrq( void *handle )
{
  ProjectXState *state = (ProjectXState *) handle;
  /* DEBUG */
  /* Hostif_ConsolePrint( state->hostif, "setIrq called. cycles: %d\n",
ARMulif_Time(&state->coredesc) ); */
  ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ, TRUE );
}

static void setFiq( void *handle )
{
  ProjectXState *state = (ProjectXState *) handle;
  /* DEBUG */
  /*Hostif_ConsolePrint( state->hostif, "setFiq called. cycles: %d\n",
ARMulif_Time(&state->coredesc) ); */
  ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_FIQ, TRUE );
}

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

#define SORDI_DLL_NAME_STRING "ProjectX"
#define SORDI_DLL_DESCRIPTION_STRING "Exception Generator model"
#define SORDI_RDI_PROCVEC ProjectX_AgentRDI
#include "perip_sordi.h"

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

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

5.3 Writing ARM application code

You need some application code to exercise this memory model.

Create a new C file, called `interrupts.c` and copy the following text into it[1]:

```
#include <stdio.h>
#include <time.h>

#define IRQNOW *intAddress = 1
#define FIQNOW *intAddress = 2
#define CLEARIRQ *clearAddress = 1
#define CLEARFIQ *clearAddress = 2
#define IRQIN(cycles) *irqAddress = cycles
#define FIQIN(cycles) *fiqAddress = cycles

unsigned *intAddress = (unsigned*) 0x200000;
unsigned *irqAddress = (unsigned*) 0x200004;
unsigned *fiqAddress = (unsigned*) 0x200008;
unsigned *clearAddress = (unsigned*) 0x20000C;

/* function to enable IRQ and FIQ */
__inline void enable_IF(void);

extern int exit(int);

unsigned Install_Handler( unsigned routine, unsigned *vector )
{
  unsigned vec, oldvec;

  vec = ((routine - (unsigned)vector - 0x8) >> 2 );
  if( vec & 0xff000000 ) {
         printf( "Installation of exception handler failed\n" );
         exit(1);
  }

  vec = 0xea000000 | vec;
  oldvec = *vector;
  *vector = vec;
  return (oldvec);
}

__irq void IRQHandler(void)
{
  CLEARIRQ;
}

__irq void FIQHandler(void)
{
  CLEARFIQ;
}

int main()
{
  unsigned *irqvec = (unsigned*) 0x18;
  unsigned *fiqvec = (unsigned*) 0x1c;

  printf( "Installing handlers\n" );
```

[1] The source code for all examples is available for download.

```
  (void)Install_Handler( (unsigned)IRQHandler, irqvec );
  (void)Install_Handler( (unsigned)FIQHandler, fiqvec );

  enable_IF();

  printf( "Handlers installed\n" );

  printf( "Cause an irq\n" );
  IRQNOW;

  printf( "Cause an FIQ\n" );
  FIQNOW;

  printf( "Cause an irq\n" );
  IRQNOW;

  IRQIN(600);
  printf( "1\n2\n3\n4\n5\n6\n7\n8\n9\n" );
  FIQIN(1500);
  printf( "1\n2\n3\n4\n5\n6\n7\n8\n9\n" );

  IRQIN(1000);
  FIQIN(1000);

  printf( "1\n2\n3\n4\n5\n6\n7\n8\n9\n" );
}

__inline void enable_IF(void)
{
  int tmp;
  __asm
  {
    MRS tmp, CPSR
    BIC tmp, tmp, #0xC0
    MSR CPSR_c, tmp
  }
}
```

Now build an ARM executable image using armcc.

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
;; NOTE: Addresses over 2GB do not read in properly
Range:Base=0x200000
}

Running the code 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

Create a new file called mycopro.c and insert the following code into it:

```
/* mycopro.c - ARM apps note 32 */
#include "minperip.h"
#include "armul_mem.h"
#include "armul_copro.h"

#define BIT(n) ( (ARMword)(instr>>(n))&1)   /* bit n of instruction */
#define BITS(m,n) ( (ARMword)(instr<<(31-(n))) >> ((31-(n))+(m)) ) /* bits m to n of
instr */
#define TOPBITS(n) (instr >> (n)) /* bits 31 to n of instr */

/*
 * What follows is a copy of the validation Suite Coprocessor.
 * It has the follwing functionality.
 * Sixteen registers.
 * co-processor can be used in an MCR and MRC to access
 *   these registers.
 * LDC and STC to and from the registers.
 * Will busy wait for the number of cycles specified by a CP register.
 * CDP 1 issues a FIQ after a number of cycles (specified in a CP
 *   register),
 * CDP 2 issues an IRQW in the same way, CDP 3 and 4 turn off the FIQ
 *   and IRQ source, and CDP 5 stores a 32 bit time value in a CP
 *   register (actually it's the total number of N, S, I, C and F
 *   cyles)

*/

/* copro regs */

static ARMword ValReg[16];

/* new prototypes follow NCAccessFunc declaration */

static int LDC( void *handle, int type, ARMword instr, uint32 *data );
static int STC( void *handle, int type, ARMword instr, uint32 *data );
static int MRC( void *handle, int type, ARMword instr, uint32 *data );
static int MCR( void *handle, int type, ARMword instr, uint32 *data );
static int CDP( void *handle, int type, ARMword instr, uint32 *data );

static unsigned DoAFIQ(void *handle);
static unsigned DoAIRQ(void *handle);

/* initialisation ************************************************/

BEGIN_STATE_DECL(Mycopro)

  /* store details of peripheral registration */
  ARMul_CoprocessorV5 cpv5;
```

```
END_STATE_DECL(Mycopro)

BEGIN_INIT(Mycopro)
Hostif_PrettyPrint(state->hostif, config, ", Mycopro");
{
  /* register the co-pro functions */
  ARMul_CoprocessorV5 *cp = &state->cpv5;
  cp->LDC=LDC;
  cp->STC=STC;
  cp->MRC=MRC;
  cp->MCR=MCR;
  cp->CDP=CDP;

  /* prototype: unsigned ARMulif_InstallCoprocessorV5(RDI_ModuleDesc *mdesc, unsigned
number,
                                     struct ARMul_CoprocessorV5 *cpv5,
                                     void *data);
    */

    ARMulif_InstallCoprocessorV5( &state->coredesc, 4, cp, state );
}
END_INIT(Mycopro)

BEGIN_EXIT(Mycopro)
END_EXIT(Mycopro)

/* COPRO instructions ****************************************/

/* copro LDC (ARM -> Copro instruction) */

static int LDC(void *handle, int type, ARMword instr, uint32 *data)
{
  static unsigned words;
  IGNORE(handle);

  if(type != ARMul_CP_DATA) {
        words = 0;
        return(ARMul_CP_DONE);
  }

  if( BIT(22) ) { /* long access so get two words */
        ValReg[BITS(12,15)] = *data;
        if( words++ == 4 )
              return(ARMul_CP_DONE);
        else
              return(ARMul_CP_INC);
  } else { /* just get 1 word */
        ValReg[BITS(12,15)] = *data;

        return(ARMul_CP_DONE);
  }
}

/* Copro. STC instruction i.e. copro -> ARM transfer */
static int STC(void *handle, int type, ARMword instr, uint32 *data)
{
  static unsigned words;

  IGNORE(handle);

  if( type != ARMul_CP_DATA ) {
        words = 0;
        return(ARMul_CP_DONE);
```

```
  }
  if( BIT(22) ) { /* two word access */
        *data = ValReg[BITS(12,15)];
        if( words++ == 4 )
              return (ARMul_CP_DONE);
        else
              return (ARMul_CP_INC);
  }
  else { /* get 1 word */
        *data = ValReg[BITS(12,15)];
        return (ARMul_CP_DONE);
  }
}

/* copro MRC instruction. copro -> ARM transfer */
static int MRC(void *handle, int type, ARMword instr, uint32 *data)
{
  IGNORE(handle);
  IGNORE(type);
  *data = ValReg[BITS(16,19)];
  return(ARMul_CP_DONE);
}

/* copro MCR instruction.  ARM to copro transfer. */
static int MCR(void *handle, int type, ARMword instr, uint32 *data)
{
  MycoproState *state = (MycoproState *)handle;

  IGNORE(handle);
  IGNORE(type);
  ValReg[BITS(16,19)] = *data;

  return (ARMul_CP_DONE);
}

/* copro CDP instruction.  ARM causes this to be executed on copro. */
static int CDP(void *handle, int type, ARMword instr, uint32 *data)
{
  static ARMTime finish;
  ARMword howlong;
  MycoproState *state=(MycoproState *)handle;
  ARMTime Now, nextEventTime;

  howlong = ValReg[BITS(0,3)];

  switch((int)BITS(20,23)) {
  case 0: if(type == ARMul_CP_FIRST) {
        /* 1st cycle of busy wait */
        finish = ARMulif_Time(&state->coredesc) + howlong;
        if( howlong == 0 )
              return ARMul_CP_DONE;
        else
              return ARMul_CP_BUSY;
        }

        return ARMul_CP_DONE;

  case 1: if( howlong == 0 )
                    ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_FIQ,
FALSE );
        else {
              Now = ARMulif_Time(&state->coredesc);
```

```
                nextEventTime = Now + howlong;

                ARMulif_ScheduleNewTimedCallback( &state->coredesc, DoAFIQ, state,
nextEventTime, 0 );
        }
        return ARMul_CP_DONE;

  case 2: if( howlong == 0 )
                      ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ,
FALSE );
        else {
                Now = ARMulif_Time(&state->coredesc);
                nextEventTime = Now + howlong;

                ARMulif_ScheduleNewTimedCallback( &state->coredesc, DoAIRQ, state,
nextEventTime, 0 );
        }
        return ARMul_CP_DONE;

  case 3: ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_FIQ, FALSE );
        return ARMul_CP_DONE;
  case 4: ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ, FALSE );
        return ARMul_CP_DONE;
  case 5: ValReg[BITS(0,3)] = (unsigned long)ARMulif_Time(&state->coredesc);
        return ARMul_CP_DONE;
  }

  return ARMul_CP_CANT;
}

/* cause an FIQ */
static unsigned DoAFIQ(void *handle)
{
  MycoproState *state=(MycoproState *)handle;
  ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_FIQ, TRUE );
  return 0;
}

/* cause an IRQ */
static unsigned DoAIRQ(void *handle)
{
  MycoproState *state=(MycoproState *)handle;
  ARMulif_SetSignal( &(state->coredesc), RDIPropID_ARMSignal_IRQ, TRUE );
  return 0;
}

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


#define SORDI_DLL_DESCRIPTION_STRING "Example Coprocessor model"
#define SORDI_RDI_PROCVEC Mycopro_AgentRDI
#include "perip_sordi.h"

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

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

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.

Create a new assembler file, called `copro.s`, and copy the following text into it:

```
        AREA Block, CODE, READONLY

        EXPORT DoCP4MCR0
        EXPORT DoCP4MRC0
        EXPORT DoCP4CDP1
        EXPORT DoCP4CDP2
        EXPORT DoCP4CDP3
        EXPORT DoCP4CDP4
        EXPORT DoCP4CDP5

DoCP4MCR0     MCR p4, 0, R0, c0, c0, 0
              MOV pc, lr

DoCP4MRC0     MRC p4, 0, R0, c0, c0, 0
              MOV pc, lr

DoCP4CDP1     CDP p4, 1, c0, c0, c0
              MOV pc, lr

DoCP4CDP2     CDP p4, 2, c0, c0, c0
              MOV pc, lr

DoCP4CDP3     CDP p4, 3, c0, c0, c0
              MOV pc, lr

DoCP4CDP4     CDP p4, 4, c0, c0, c0
              MOV pc, lr

DoCP4CDP5     CDP p4, 5, c1, c1, c1
              MRC p4, 0, a1, c1, c1, 0
              MOV pc, lr

              END
```

Create another new C file, called `coprotest.c`, and copy the following text into it:

```
#include <stdio.h>
#include <stdlib.h>
```

```
/* exported functions in copro.s */
extern void DoCP4MCR0(unsigned);
extern unsigned DoCP4MRC0(void);
extern void DoCP4CDP0(void);
extern void DoCP4CDP1(void);
extern void DoCP4CDP2(void);
extern void DoCP4CDP3(void);
extern void DoCP4CDP4(void);
extern unsigned int DoCP4CDP5(void);


/* function to enable IRQ and FIQ */
__inline void enable_IF(void);

/* exception handler install */

unsigned Install_Handler( unsigned routine, unsigned *vector )
{
  unsigned vec, oldvec;

  vec = ((routine - (unsigned)vector - 0x8) >> 2 );
  if( vec & 0xff000000 ) {
        printf( "Installation of exception handler failed\n" );
        exit(1);
  }

  vec = 0xea000000 | vec;
  oldvec = *vector;
  *vector = vec;
  return (oldvec);
}

__irq void IRQHandler(void)
{
  printf( "IRQ entered\n" );
  DoCP4CDP4();
}

__irq void FIQHandler(void)
{
  printf( "FIQ entered - clear source\n" );
  DoCP4MCR0(0); /* put 0 in CP4 c0 */
  DoCP4CDP1(); /* Use CDP 1 with time = 0 to clear irq source */
}

int main()
{
  unsigned *irqvec = (unsigned*)0x18;
  unsigned *fiqvec = (unsigned*)0x1c;
  unsigned value=0;
  unsigned int timerValue=0;

  printf( "Installing handlers\n" );

  Install_Handler((unsigned)IRQHandler, irqvec);
  Install_Handler((unsigned)FIQHandler, fiqvec);

  enable_IF();

  printf("Handlers installed\n");

  printf("Use MRC/MCR \n" );
  DoCP4MCR0(0xFFFFFFFF); /* put 0xFFFFFFFF in CP4 reg 0 */

  value = DoCP4MRC0(); /* get above value from CP4 reg 0 */
  printf( "Value = %08x\n", value );
```

```
  DoCP4MCR0(0xaaaaaaaa); /* put 0xaaaaaaaa in CP4 reg 0 */
  value = DoCP4MRC0();
  printf("Value = %08x\n", value);

  printf( "CDP 1 - cause a FIQ\n" );
  DoCP4MCR0(500);
  DoCP4CDP1();
  printf( "Expecting a fiq. \nCDP1 done\n\n\n");

  printf( "CDP 2 - cause an IRQ\n" );
  DoCP4MCR0(500);
  DoCP4CDP2();
  printf( "Expecting an irq. \nCDP2 done\n" );

  printf( "\n\nRead timer\n" );

  timerValue = DoCP4CDP5();
  printf( "Value = %08x\n", timerValue );
  timerValue = DoCP4CDP5();
  printf( "Value = %08x\n", timerValue );

}

__inline void enable_IF(void)
{
  int tmp;
  __asm
  {
    MRS tmp, CPSR
    BIC tmp, tmp, #0xC0
    MSR CPSR_c, tmp
  }
}
```

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 = aaaaaaaa
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. 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. (See AppNote 32 for more information)

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 as per the examples in the AppNote. This means that a subdirectory exists containing your model and a corresponding makefile, named MyModel.b.

- 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 settings

- Open the settings dialog by choosing the menu Project->Settings… or pressing ALT+F7.
- The drop-down list at the top-left should display Win32 Debug by default. If not, select this option.
- On the Debug tab, choose category “General” from the drop-down list and select a file for the “Executable for debug session”. Choose install_dir\Bin\axd.exe or another debugger of your choice. install_dir refers to the directory in which you installed ADS 1.1.
- On the C/C++ tab, choose “General” from the drop-down menu and change /MTd in the “Project Options” box to /MD. Next, choose “Preprocessor” from the drop-

down menu and specify the following in “Additional include directories”: ..\..\rdi,..\..\clx,..\..\armulif

- On the Link tab, choose “General” and replace the contents of the Object/library modules box with: ..\..\clx\clx.b\intelrel\clx.lib ..\..\armulif\armulif.b\intelrel\armulif.lib. In the ‘output file name’ box, enter install_path\bin\mymodel.dll.

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 install_dir\bin directory.

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 under ADS 1.1. A simpler mechanism will be available under ADS 1.2.

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:

```
/* 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 Appendix A – Known changes required for ADS 1.2

- When creating a .dsc peripheral configuration file, ADS 1.2 uses `MODEL_DLL_FILENAME` instead of `META_SORDI_DLL` to specify the library file to use. See section 3.1.2 for more details.
- The structure of the ARMulator has been altered slightly to allow the flat memory model and decoder to be modified. This also exposes part of the ARMulator API (Application Programming Interface) to allow simple insertion of peripheral callbacks which occur every cycle. The flat memory model will be available in the file `flatmem.c`.