OVP Peripheral Modeling Guide

A guide to writing behavioral components / peripheral models in the OVP and Imperas environments.

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas
Version:	1.3.3
Filename:	OVP_Peripheral_Modeling_Guide.doc
Last Saved:	Wednesday, 22 July 2015
Keywords:	Peripheral PSE Modeling OVP

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Preface

This document describes the features available for modeling peripheral components using the OVP and Imperas tools.

1.1 Notation

`Code`	Text representing a code extract.
keyword	A word with special meaning.

Note that for clarity, examples generally omit error handling code.

1.2 Related OVP Documents

- BHM PPM Function Reference

2 Introduction

OVP and Imperas simulation technology enables very high performance simulation, debug and analysis of platforms containing multiple processors and peripheral models. The technology is designed to be extensible: you can create new models of processors, peripherals and other platform components using interfaces and libraries supplied by Imperas.

This document describes how to use the OVP interfaces to create your models. Specifically, it covers implementation of:

- peripheral models;
- peripheral interception libraries (which enable peripheral models to interact with the native host);

The documentation here is supported by C code samples in the `Examples` directory of your OVP / Imperas installation.

2.1 Prerequisites

Since models for use with OVP and Imperas tools are written in C, an important prerequisite is that you must be an expert in the C language.

GCC Compiler Versions

Linux32	4.5.2	i686-nptl-linux-gnu (Crosstool-ng)
Linux64	4.4.3	x86_64-unknown-linux-gnu (Crosstool-ng)
Windows32	4.4.7	mingw-w32-bin_i686-mingw
Windows64	4.4.7	mingw-w64-bin_i686-mingw

2.1.1 MinGW

Note that the MinGW program ***make*** is suitable for use with the PSE tool-chain and will compile the examples in this document.

2.1.2 PSE Toolchain

To compile and link a peripheral model, you will need the Imperas Peripheral Simulation Engine (PSE) tool-chain, which can be downloaded from the OVP website (www.ovpworld.com). This is based on GNU gcc and comprises a C-compiler, assembler, linker, objdump and other utilities.

2.2 Modeling Data Endianness

The Peripheral models created using the OVP APIs run on the Peripheral Simulation Engine (PSE) and are always implemented using the little endian data format, as they run natively on the host x86 platform.
To model a big endian format peripheral device the data format must be converted at the interface between the peripheral model and the platform.

3 OVP Simulation Overview

Before starting to create models for use with the OVP simulation environment, you must understand how the components used in that environment interact. This section describes this in detail.

3.1 Imperas & OVP Tools

There are several Imperas and OVP tools that can be used with models that you create:

- *Imperas OVPsim* allows processor models created using OVP modeling technology to be used in C harness or platform files to create executables that execute binaries compiled for those processor models. It can also simulate behavioral components (the subject of this guide). OVPsim can also be used in 3rd party simulation environments (for example, SystemC). It can also be used to create a test harness to help validate processor models under construction, or even to create a custom simulation environments. OVPsim has less functionality than the Imperas CpuManager product.
- *Imperas CpuManager* allows processor models created using OVP modeling technology to be used in 3rd party simulation environments (for example, SystemC). It can also be used to create a test harness to help validate processor models under construction, or even to create a custom simulation environment. It can also be used to create standalone virtual platform executables.
- *Imperas Explorer* provides the generic simulation framework for multiprocessor simulation. It reads XML-format *platform files* that describe arbitrary multiprocessor systems. To construct the simulation models of those systems, it loads shared objects (or dynamic linked libraries on Windows) which implement the behavior of the platform components.

3.2 Processor Models

The core simulation components are *processor models*. The creation of a new processor model is described in detail in the documentation “Imperas Processor Modeling Guide” with reference to the OVP *Virtual Machine Interface* (VMI) API.

3.3 Peripheral Models

The creation of a new peripheral models is described in this document with reference to the OVP BeHavioral Modeling (BHM) and the Peripheral Programming Model (PPM) APIs.

A peripheral model is compiled into an ELF format executable for the PSE processor architecture. It is dynamically loaded by OVPsim or other tools. If the peripheral model is to be used by the Imperas Explorer simulator then an XML file also needs to be created to describe the interface to the PSE executable.

3.4 Intercept Library

The PSE processor runs in it own private (simulated) memory space, which is isolated from the host environment. *Interception* of functions defined in the peripheral model allows the use of features of the host system in the implementation of the behavior of a peripheral.

As an example, a real platform might contain a video display device. When simulating this system, it is generally more convenient not to simulate the complete video display device but to use a video package available on the host machine, such as SDL, and to use this to render to the host display.

Using OVP & Imperas technology, interception libraries can be created as loadable shared objects (or dynamic linked libraries on Windows).

Implementation of an interception library is described in Native Host Code for Functional Behavior in section 7.

4 Behavioral Modeling Methodology

Each instance of a peripheral model runs on its own virtual machine with an address space large enough for the model. This processor and its memory are separate from any processors, memories and buses in the platform being simulated; they exist only to execute the code of the peripheral model. This processor is called a Peripheral Simulation Engine or PSE for short.

The model writer can use normal C constructs, and some libc library functions:

- some low level i/o:
  - __open, close, read, write, fstat, lstat, lseek, unlink,
  - gettimeofday, exit, time, times.
- libc I/O built on the above
  - fopen, fread, fwrite etc.
- functions which do not use kernel functions:
  - string handling
  - searching and sorting
  - mathematics functions

⇒ The use of networking, graphics or other I/O routines is prohibited

The libc file handling functions give access to the host file system.

Additionally, models can create and control threads of execution using the BHM API and can interact with other components in the platform using the PPM API.
The code is not preempted; while peripheral code is executing, the simulator does not advance simulated time, and no other activity occurs in the platform.

OVPsim provides a simulation environment which is accurate enough to verify the functionality and performance of a platform that typically contains both processors running application software and peripherals.

Each processor is provided as a separate model and peripherals are modeled in C and run on a Peripheral Simulation Engine (PSE).

This section is intended to provide an example of a peripheral model using functions provided within the OVP BeHavioral programming Model (BHM) C-API and Peripheral Programming Model (PPM) C-API. The document uses the OVP virtual platform modeling environment.

The BHM provides the ability to write behavioral models of hardware and to write testbench code which interacts with other models. To do this a task-based programming model allows the user to:

- Create and delete tasks
- Let a task wait for an amount of simulated time
- Let a task wait for an event in another task
- Perform basic host OS I/O functions

5 Behavioral Modeling (BHM) API Overview

This section introduces the features available to model the behavior of a peripheral. These features will be used in a later section in an example that creates a DMA controller.

All peripheral models use the BeHavioral Modeling (BHM) API. This is included by a single header file

```
#include “bhm.h”
```

This gives access to

- Threads
- Events and named events
- Simulated delays
- Simulator control
- The simulator message stream and diagnostics control.

5.1 Definition

A peripheral model must provide a structure describing its interface, which can be interrogated by the simulator before any peripheral model code is executed. The structure must be called `“modelAttrs”` and use the type “`ppmModelAttr”`. The following example shows how to define bus-ports, net-ports and model parameters.

```
static ppmBusPort busPorts[] = {
    {
        .name            = "DMACSP",
        .type            = PPM_SLAVE_PORT,
        .addrHi          = 0x13fLL,
        .mustBeConnected = 1,
        .description     = "DMA Registers Slave Port",
    },
    {
        .name            = "MREAD",
        .type            = PPM_MASTER_PORT,
        .addrBits        = 32,
        .description     = "DMA Master Port ",
    },
    { 0 }
};

static PPM_BUS_PORT_FN(nextBusPort) {
    if(!busPort) {
        busPort = busPorts;
    } else {
        busPort++;
    }
    return busPort->name ? busPort : 0;
}
```

```
static PPM_NET_CB(netChangeNotifier) {
    …
}

static ppmNetPort netPorts[] = {
    {
        .name            = "INTTC",
        .type            = PPM_OUTPUT_PORT,
        .description     = "Interrupt Request"
    },
    {
        .name            = "DRQ",
        .type            = PPM_INPUT_PORT,
        .mustBeConnected = 1,
        .description     = "Data Request",
        .netCB           = netChangeNotify
    },
    { 0 }
};

static PPM_NET_PORT_FN(nextNetPort) {
    if(!netPort) {
        netPort = netPorts;
    } else {
        netPort++;
    }
    return netPort->name ? netPort : 0;
}

////////////////////////////// PACKETNET ////////////////////////////////////

Typedef struct myDataPacketS {
    ...
} myDataPacket;

static PPM_PACKETNET_CB(pktTrigger) {
    ...
}

static ppmPacketnetHandle pktPortHandle;

static ppmPacketnetPort packetnetPorts[] = {
    {
        .name            = "pktPort",
        .description     = "Packetnet port",
        .sharedData      = pktPortPsd,
        .sharedDataBytes = sizeof(myDataPacket),
        .handlePtr       = &pktPortHandle,
        .packetnetCB     = pktTrigger,
    },
    { 0 }
};

static PPM_PACKETNET_PORT_FN(nextPacketnetPort) {
    if(!packetnetPort) {
        packetnetPort = packetnetPorts;
    } else {
        packetnetPort++;
    }
    return packetnetPort ->name ? packetnetPort: 0;
}
```

```
static ppmParameter parameters[] = {
    {
        .name         = “MAX_BURST”,
        .type         = ppm_PT_UNS32,
        .description  = “Maximum DMA bust length”,
        .u.uns32Param = { 100, 10, 1000 }
    }
    { 0 }
};

static PPM_PARAMETER_FN(nextParameter) {
    if(!parameter) {
        return parameters;
    }
    parameter++;
    return parameter->name ? parameter : 0;
}

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,

    .busPortsCB       = nextBusPort,
    .netPortsCB       = nextNetPort,
    .packetnetPortsCB = nextPacketnetPort,
    .paramSpecCB      = nextParameter

};
```

5.1.1 Bus port definitions

`ppmBusPort` is a structure filled by the model and read by the simulator. The function `nextBusPort` is a callback which should return a pointer to each `ppmBusPort` structure, ending with null. Each returned structure describes one bus port.
This example uses a static array for simplicity. A more complex model could construct the structures on the fly.

5.1.2 Net port definitions

`ppmNetPort` is a structure filled by the model and read by the simulator. The function `nextNetPort` is a callback which should return a pointer to each `ppmNetPort` structure, ending with null. Each returned structure describes one net port.
This example uses a static array for simplicity. A more complex model could construct the structures on the fly. The first net port is an output, the second an input which calls the model’s function `netChangeNotify.`

5.1.3 Packetnet port definitions

`ppmPacketnetPort` is a structure filled by the model and read by the simulator. The function `nextPacketnetPort` is a callback which should return a pointer to each `ppmPacketnetPort` structure in turn, ending with null. Each returned structure describes one packetnet port.
This example uses a static array. A more complex model could construct the structures on the fly. Packetnet ports can be bidirectional but a model might choose to use the port unidirectionally.

The simulator sets the handle `pktPortHandle` which can then be used by the function `ppmPacketnetWrite()` to transmit data:

```
...
myDataPacket pkt = { ... };
ppmPacketnetWrite(pktPortHandle, &pkt, sizeof(pkt));
...
```

The function `pktTrigger` will be called when a packet arrives.

The maximum number of bytes supported by the protocol is specified in the sharedDataBytes field in the packetnetPort structure. This value must match the corresponding size in all models connected to the packetnet.

A complete example using a packetnet is in:

```
$IMPERAS_HOME/Examples/Models/Peripherals/packetnet
```

5.1.4 Parameter definitions

`ppmParameter` is a structure filled by the model and read by the simulator. The function `nextParameter` is a callback which should return a pointer to each `ppmParameter` structure, ending with null. Each returned structure describes one parameter.
This example uses a static array for simplicity. A more complex model could construct the structures on the fly. A single unsigned 32bit parameter has a default value of 100 and can be set to values from 10 to 1000.

5.2 Initialization

A peripheral model must declare a function `main()`.
At the start of simulation this will be called (with null `argc` and `argv`). It should

- perform software initialization.
- make any connections to the hardware platform (see Peripheral Platform Modeling (PPM) API Overview in section 6).
- perform any hardware reset functions to initialize the peripheral[1].
- start any threads required by the model. If only one thread is required, this can be written as a loop in the main thread.
- optionally, wait for the ‘end of simulation’ event (see System Events in section 5.3.2).

```
int main(int argc, char *argv[])
{
    busPortConnections()
    netPortConnections();
    userInit();
```

[1] If the model is programmed to respond to a reset input port, this action will also be in a callback.

```
    bhmWaitEvent( bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION) );
    return 0;
}
```

5.3 Threads and Events

Peripheral models can use a light-weight cooperative threading model to express concurrent behavior. An example might be a multi-channel DMA engine where each channel can be in one of a number of states independent of the other channels.

A thread is started using the `bhmCreateThread()` function.

A thread requires a stack which must be allocated by the user. The stack should have sufficient space for that thread and any code it uses[2]. The stack grows down and as such the address of the TOP of the stack is passed to the peripheral thread creation routine.

A thread is given a name and can receive a user-defined value, typically used if several copies of the same thread are launched with different contexts.

```
#include “bhm.h”

bhmThreadHandle thA, thB;  // only required if you wish to delete the thread

#define size (64*1024)

char stackA[size];
char stackB[size];

struct myThreadContext contextA;
struct myThreadContext contextB;

void myThread(void *user)
{
    struct myThreadContext *p = user;
    while(1) {
        bhmWaitDelay(1000*1000);
        bhmPrintf(“tick\n”);
        bhmWaitDelay (1000*1000);
        bhmPrintf(“tock\n”);
    }
}

void userInit(void)
{
    thA = bhmCreateThread(myThread, &contextA, “threadA”, &stackA[size]);
    thB = bhmCreateThread(myThread, &contextB, “threadB”, &stackB[size]);
}
```

[2] Be aware that libc can use a significant amount of stack

Once started, a thread will run to the exclusion of all other simulator activity until a wait of some kind is executed. Therefore a thread's main loop must include a wait. Calls which wait are:

- `bhmRestartReason bhmWaitEvent(bhmEventHandle h);`
- `bhmWaitDelay(double microseconds);`

Threads can be created (bhmCreateThread) at any time.
They can be destroyed from another thread (bhmDeleteThread) or can destroy themselves by returning from the thread entry function.

Although threads are visible to the debugger, they cannot be seen by models outside this PSE.

5.3.1 Events

Multiple threads are synchronized using events. An event handle is declared, an event is created and can then be used to stop a thread until another thread or callback triggers it.

```
bhmEventHandle  go_eh;

void myThread(void *user)
{
    while(1) {
        bhmWaitEvent(go_eh);

        … etc …
    }
}

void userInit(void)
{
    go_eh = bhmCreateNamedEvent("start", "start a transaction");
    bhmCreateThread(myThread, userData, “myThread”, &stack[size]);
}

void otherThread(void *user)
{
    … etc …

    bhmTriggerEvent(go_eh);

    … etc …
}
```

5.3.2 System Events

Two system events can be accessed using `bhmGetSystemEvent()`.

- `BHM_SE_START_OF_SIMULATION`
- `BHM_SE_END_OF_SIMULATION`

`The event BHM_SE_START_OF_SIMULATION` is triggered, by the simulator, after all the peripherals in the platform have executed code to their first wait. No application processors have started.

The event `BHM_SE_END_OF_SIMULATION` is triggered, by the simulator, after all the application processors have finished. This can be used to terminate the peripheral model when the application processors have completed execution. This may be something causing the simulation to terminate prematurely.

```
bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
```

The peripheral model should never trigger one of the system events.

5.3.3 Delays

A task can wait for a simulated delay or trigger an event in the future without waiting.

An outstanding trigger can be cancelled. Only one trigger can be outstanding on an event; re-issuing a trigger will cancel any other outstanding event.

```
bhmEventHandle  ev1, ev2;

void thread1(void *user)
{
    while(1) {
        bhmWaitDelay(120 /*uS*/);
        bhmTriggerEvent(ev1);
    }
}

void thread2(void *user)
{
    while(1) {
        bhmWaitEvent(ev1);
        bhmTriggerAfter(ev2, 120 /*uS*/);
    }
}
```

5.4 Time

A Peripheral can inquire the current simulation time. Simulation time starts at zero and will progress faster or slower than real (wall-clock) time. Stopping the simulator in a debugger will stop time. In peripheral code, time will not progress until the model makes a call to bhmWaitDelay

```
Uns64 microseconds1 = bhmGetCurrentTime();
bhmPrintf(message);

Uns64 microseconds2 = bhmGetCurrentTime();

// in all circumstances microseconds1 == microseconds2
bhmWaitEvent(ev99);
Uns64 microseconds3 = bhmGetCurrentTime();

// microseconds3 might be greater than microseconds2
```

5.5 Diagnostic output

A model should produce diagnostic output according to the model user's requirements. The user controls the output by setting the item `diagnosticlevel` in the peripheral instance in the hardware design, or by issuing the *diagnostics* command in the debugger. This triggers a call to the function registered by bhmInstalldiagCB(). The function should set a local variable which is used to control diagnostic output.
Four levels should be used to determine the required diagnostic output:

0: No output
1: Print at startup (and possibly shutdown).
2: Print change of mode, major operations etc.
3: Print maximum detail.

4: (reserved) This level causes the simulator to log when it interacts with the model; e.g. when registers are read or written, when input nets change and when events are triggered.

```
#include "bhm.h"

int diagLevel = 0;

static void setDiags(Uns32 v)
{
    diagLevel = v;
}

int main()
{
    bhmSetDiagnosticCB(setDiags);

    if (diagLevel > 0) {
        bhmMessage("I", "MY_PERIPH", "Starting up...");
    }
    if (diagLevel > 2) {
        bhmPrintf("Table heading\n");
        bhmPrintf("     %d\n", entry[1]);

    }
}
```

The routines `bhmPrintf` and `bhmMessage` send text to the simulator output stream and log file. `bhmMessage` prefixes each message with the severity, a user code and the peripheral instance name. `bhmPrintf` does not, so should be used to produce formatted tables, for example.

Note that the user can change diagnostic level during a simulation.

Note that *diagnostic* output is intended to help the user of a complete model.

5.6 Controlling a model from the platform.

It is a common requirement to be able to change the behavior of a model without changing the executable code. Applications include:

1. Implementing variants of a model without duplicating common code.
2. Choosing model options during instantiation (like strapping inputs when wiring up a device).

The BHM API allows a model to access string and integer values specified in the instantiation of the model in the platform.

```
char buffer[length];
if (bhmStringAttribute("myStrAttribute", buffer, length)) {
  … code …
}
Uns32 val;
if (bhmIntegerAttribute("myIntAttribute", &val)) {
  … code …
}
```

The calls in the example will return non-zero if myStrAttribute and myIntAttribute are user-defined attributes in the platform. A call to `bhmStringAttribute` (or `bhmIntegerAttribute`) searches first in the model definition (useful in application 1 defined above), then in the model instance and sequentially up the platform hierarchy (useful in application 2 defined above) for an attribute of that name.

5.6.1 Accessing Platform Attributes

Note that the interface also searches for built-in items and can include a relative path. This example returns the 32-bit value of the loaddress on the bus slave port connection "P1".

```
Uns32 lo;
if (bhmIntegerAttribute("P1/loaddress", &lo)) {
   … code …
}
```

6 Peripheral Platform Modeling (PPM) API Overview

Use `ppm.h` to access to the platform hardware:

The PPM provides extensions to the BHM to allow the user to interact with other components in a system platform. This may include operations to:

- Read and write memory regions
- Create and receive interrupts
- Communicate between tasks

Most peripheral models use the Peripheral Platform Modeling (PPM) API. This is included by a single header file

```
#include “ppm.h”
```

This gives access to

- Connectivity of peripherals in platforms
- Creation and control of
  - ports and nets
  - address spaces
  - windows into memory address space
- Create behavior on memory region accesses
  - Install callbacks
- Create ‘programmers view’ with
  - objects
  - events
  - actions

To interact with other components, the model is connected to the platform by its ports.

6.1 Net connection

The model connects to a net port using ppmNetPort entries in the attributes table (see section 5.1.2). The ppmNetHandle is updated by the simulator during construction. The handle can then be used to read from and write to the net.

The handle value is used to drive the net (if the port is an output):

```
ppmWriteNet(nh, newValue);
```

The value of a net can be found at any time:

```
if ( ppmReadNet(nh) == SOME_VALUE)
{
    … code …
}
```

6.2 Packetnet connection

There is a comprehensive description of packetnets in the OVPsim and CpuManager User Guide.

The model connects to a packetnet port using a callback specified in the model attributes table (see section 5.1.3). Packets are sent using ppmPacketnetWrite:

```
ppmPacketnetWrite(ppmPacketnetHandle ch, void *data, Uns32 bytes);
```

This function uses the connectivity specified in the platform to identify each destination model and port, then for each one, copies the specified number of bytes (which must not exceed the maximum number of bytes specified in the attributes table) to the address space of the destination model, then calls the model's notification function. Notification functions may modify the data contents if required. ppmPacketnetWrite returns after all models have been notified. The model can examine the data for changes made by receiving models.

To avoid infinite recursion, the packetnet notification function should not call ppmPacketnetWrite to this or other packetnets.

Data passed through a packetnet will be copied to different locations in different address spaces, so must not contain address references.

6.3 Bus Slave connection

The model connects to a bus as a slave by mapping an area of local memory to the address space of the simulated bus. The two regions must be of the same size.

6.3.1 Fixed Mapping

Using this method, the address of the slave port comes from the platform, not this model. The address offset specified as `loaddress` in the busslaveportconnection is not visible through this API; an access to the lowest address on the simulated bus appears at lowest address in the mapped region.

```
unsigned char mappedRegion[sizeInBytes];   // a region to be read/written
const char *portName = "p1";               // name matches model spec.

ppmBusHandle bh = ppmOpenSlaveBusPort(
    portName,
    mappedRegion,
```

```
    sizeof(mappedRegion)
);
```

Reads and writes by simulated processors to the region specified in the busslaveportconnection will simply read and write to the array `mappedRegion`. The peripheral model can examine or update these values as required.

If the peripheral model is required to act on a read or write (i.e. event-driven), it can install callbacks on an area of local memory. It is recommended that callbacks are installed only on externally mapped regions; if a model generates callbacks on its own reads or writes it can lead to unexpected behavior.

```
PPM_READ_CB(readCallback) {
     ...
     return valueToBeReturned;
}

PPM_WRITE_CB(writeCallback) {
     ...
}

ppmInstallReadCallback (readCallback,  NULL, mappedRegion, sizeInBytes);
ppmInstallWriteCallback(writeCallback, NULL, mappedRegion, sizeInBytes);
```

In this state, a read or write by a simulated processor to the region specified in the busslaveportconnection will cause calls to `readCallback` or `writeCallback`. The value returned by the read will be that returned by `readCallback`.

6.3.2 Dynamic Mapping

Also known as remapping, this style allows the model to specify its own address on the simulated bus, and to change the mapping as required. Note that attempting to specify an address range with overlaps with another fixed or dynamically mapped region will cause a run-time error.

```
unsigned char remappedRegion[sizeInBytes];   // area to be read/written
const char     *portName = "dp1";
static SimAddr loAddr    = initialAddress(); // remember port addr

ppmCreateDynamicSlaveBusPort(  // set the initial port address
    portName,
    loAddr,
    remappedRegion,
    sizeInBytes
);

...

ppmDeleteDynamicSlavePort(       // remove the old mapping
    portName,
    loAddr,
    sizeInBytes
);

loAddr = newAddress();            // recalculate

ppmCreateDynamicSlaveBusPort(     // remap
```

```
    portName,
    loAddr,
    remappedRegion,
    sizeInBytes
);
```

6.4 Bus master connection

There are two methods of connecting a model to its bus master port. In the simplest, a handle to the address space on the simulated bus is obtained, and then `ppmReadAddressSpace` and `ppmWriteAddressSpace` are called to cause bus cycles as required. This method gives access to up to 64 bits of address space (the maximum supported by the simulator) and also allows the simulator to see each bus cycle and so account for the peripheral's use of bus and other resources.

```
ppmAddressSpaceHandle mh = ppmOpenAddresSpace(portName);

// 32 bit read and write

Uns32 data;

// this is a read-modify-write operation.
ppmReadAddressSpace (mh, address, sizeof(data), &data);
data++;
ppmWriteAddressSpace(mh, address, sizeof(data), &data);
```

The alternative method is to map an area of local memory to the simulated bus. This allows the model to directly read and write to the simulated address space. Access to the bus is more efficient, since no API call is required but there are two disadvantages:

- The simulator cannot track bus activity caused by the model.
- It might be impossible to make the window as large as the address space to which the port is connected (because the window must be a reserved area in the peripheral's own address space). In this case, a small window must be declared and its location in the address space moved to cover the area where access is required.

```
#include "ppm.h"

static char window[size];

ppmAddressSpaceHandle mh = ppmOpenMasterBusPort(
    portName,
    window,
    size,
    remoteAddress);

// write zeros into simulated address space
memset(window, 0 , size);

// move the window along
remoteAddress += size;
ppmChangeRemoteLoAddress(mh, remoteAddress);

// write more zeros into simulated address space
memset(window, 0 , size);
```

Note that declaring a large static array for use as a window will make the model's executable large and hence slow to load.

6.5 Dynamic Bridges

A common platform requirement is for address maps to change at run-time (e.g. a PCI bus model). In this situation, the platform model describes the topology of the buses without specifying address decodes. The PSE model then maps (or remaps) address regions from one bus to another using a Dynamic Bridge. Dynamic Bridges are unidirectional; reads and writes to a bus connected to the peripheral slave port are mapped to a (possibly) different address on a bus connected to the peripheral master port. A peripheral supporting Dynamic Bridges can bridge more than one master and/or slave port at the same time and can have multiple bridges though each port. However, it is an error to create overlapping slave regions, or to overlap with other fixed ports on the same bus.

Overlapping master regions creates the effect of dual-port or shared devices. The recommended procedure for remapping is for the PSE code to keep track of active bridges and to delete them before new ones are created.

The functions ppmCreateDynamicBridge and ppmDeleteDynamicBridge implement dynamic bridging.

The first part of this example creates a region of 0x1000 bytes starting at address 0x40000000 on the bus connected to 'slavePort'. Reads or writes by bus masters on this bus to this region will be mapped to the bus connected to 'masterPort', starting at address 0. The second part removes the mapping, after which reads or writes to this area will cause a bus error.

```
#include "ppm.h"

... {
    // create bridge
    slavePortLoAddress  = 0x40000000;
    windowSizeInBytes   = 0x1000;
    masterPortLoAddress = 0;

    Bool ok = ppmCreateDynamicBridge(
        "slavePort",
        slavePortLoAddress,
       windowSizeInBytes,
        "masterPort",
        masterPortLoAddress
    );

}

...{
    // delete bridge
    ppmDeleteDynamicBridge(
        "slavePort",
        slavePortLoAddress,
       windowSizeInBytes
```

```
    );
}
```

6.6 Programmers View Overview

The programmers view is for use with Imperas Professional tools, please contact Imperas at info@imperas.com for more information.

A programmers view is created within the peripheral model to be used with the Imperas MP debugger and interception plugins[3] in order to:

1. provide a view of objects within the model, for example registers
2. define special actions that the model can perform, for example flush buffers, reset, generate test data
3. allow values to be read from objects without side effects on the model
4. generate **eventpoints** (a break on an event being triggered) on specific condition being met, for example data received, buffer overflow, buffer empty

6.6.1 Automatic Object and Event Generation

Some ppm API commands automatically generate information for the programmers view. Creating registers using the *ppmCreateRegister* or *ppmCreateInternalRegister* functions creates both a register object that can be accessed by the Imperas MP Debugger and also read and write events that can trigger the Imperas MP Debugger and Intercept Plugins. See section 8.5.3 Register Behavior for more information.

6.6.2 Objects

6.6.2.1 Introduction

An object can be added using the *ppmAddViewObject* function as a hierarchy of objects.

The initial object will be supplied with a parent of *NULL* indicating that it is the top-level object, subsequent objects can be added within previously added objects to allow hierarchy, for example fields within a register.

When an object is read the value provided may be passed from the peripheral model as either a

1. variable
2. constant value
3. function call

The types of values that can be associated with an object are defined in the *ppmViewValueType* enumerated type in the ppm header file.

6.6.2.2 Creating an Object

In the following example a new top level object *viewRegisters* is created that contains a register *control*

[3] See the "Imperas Interception Plugins User Guide" for more information.

```
//
// Create tree of view objects.
// A register block with a read-only counter register and a read/write control
register.
//
ppmViewObject viewRegisters = ppmAddViewObject(
        NULL,                           // Parent view object.
                                        // NULL to add object at top-level.
        "registers",                    // Object name
        NULL                            // Optional description string
    );

// Control register and fields.
ppmViewObject viewControlReg = ppmAddViewObject(
        viewRegisters,
        "control",
        NULL
);
```

6.6.2.3 Associating Values with Objects

The value of the register *control* is accessed using a callback function so is defined using the function `ppmSetViewObjectValueCallback`.

```
ppmSetViewObjectValueCallback(viewControlReg, readControlValueCB, 0);
```

We can add further registers and also bit fields within registers. The programmers view allows the register to be viewed and accessed in any way the peripheral model creator wishes them to be available.

The following code snippet shows the creation of an object for an interrupt enable bit within the control register that can be separately accessed. A new object is created from the parent control register object and a reference value associated of an internal model variable that reflects the interrupt enable bit state.

```
ppmViewObject viewIEN = ppmAddViewObject(
        viewControlReg,
        "ien",
        "interrupt enabled (bit 7)"
);
ppmSetViewObjectRefValue(viewIEN, PPM_VVT_BOOL, &interruptsEnabled);
```

A further object is created for another field from the control register.

```
ppmViewObject viewRATE = ppmAddViewObject(
        viewControlReg,
        "rate",
        "counter tick rate in cycles per tick (bits 5:0)"
);
ppmSetViewObjectRefValue(viewRATE, PPM_VVT_UNS8, &tickRate);
```

6.6.2.4 Removing an Object

An object may be transient i.e. it is not always valid. An object can be removed at any time using the `ppmDeleteViewObject` function.

```
ppmDeleteViewObject(thisObject);
```

6.6.3 Events

6.6.3.1 BHM Events

A BHM event is mainly used for signaling between threads in a peripheral model or so a peripheral model can respond to system events.

The *bhmCreateNamedEvent* function is used to create a named event that can be used between threads but also is available to the Imperas professional tools to be used for an eventpoint.

6.6.3.2 Adding an Event

An event can be added into the peripheral model to allow anything to be signaled to the Imperas MP Debugger or an intercept plugin. An event can be generated at the top-level or it can be associated with an object declared in a peripheral model using the *ppmAddViewEvent* function. Where an event is declared does not change its behavior or how it is used but allows events to be reported associated with an object.

The following shows two examples of events, one being created at the top level and one associated with an object.

```
//
// Create events which can trigger eventpoints
//
interruptEvent = ppmAddViewEvent(
        NULL,                               // Parent view object.
                                            // NULL means add event at top-level
        "interrupt",                        // Event name
        "generated timer interrupt"         // Event description. Can be NULL.
    );

wrapEvent = ppmAddViewEvent(
        viewCounterReg,
        "wrap",
        "triggered when timer counter wraps"
);
```

6.6.3.3 Triggering an Event

An event is specifically triggered in a peripheral model using the *ppmTriggerViewEvent* function call at any point in the behavior of the peripheral model. As shown in the following code snippet taken from the behavior of the counter register in the peripheral model. When the counter has wrapped an interrupt function is called, to provide interrupt behavior, and the *wrapEvent* is triggered. Anything waiting on or having a breakpoint set on this event will be triggered.

```
  if ( counter == 0 ) {
```

```
    // Counter just wrapped. Generate interrupt if enabled.
    generateInterrupt();

    // Inform simulator of wrap.
    ppmTriggerViewEvent(wrapEvent);
  }
```

6.6.4 Actions

Arbitrary actions can be created in a peripheral model. These are not limited to modifying register and other variable values but can be used to perform arbitrary functionality. An action is registered using the *ppmAddViewAction* function along with a function that performs the action.

In the following code an action is created that will cause the counter from our example to be reset.

A function is used to create the required behavior

```
//
// Action callbacks invoked when simulator/debugger wants to perform an action.
// Change model state.
//
void resetCounterActionCB(void *userData) {
    resetCounter();
}
```

The function is then attached to an action that is available to the debugger.

```
  ppmAddViewAction(
        viewCounterReg,              // Parent view object. Counter register.
        "reset",
        "reset the timer counter",
        resetCounterActionCB,
        0
  );
```

7 Native Host Code for Functional Behavior

7.1 Introduction

The PSE provides an isolated, protected, threaded environment to run a peripheral model. Access to the host environment is necessarily restricted; system functions open, close, read, write and stat (and their libc buffered equivalents fopen() etc.) are all that is available (supported by *semihosting*). Should a model require access to other system functions, the user can use the technique of *Interception*, in particular *Function Interception* to link to functionality provided on the native host machine.

There is a comprehensive description of function interception applied to application code in the "OVP VMI OS Support Function Reference" document. This section aims to illustrate the use of interception to enable the link between peripheral model behavior provided by a PSE and model functionality provided by native host code.

Function interception allows a peripheral model to be created that comprises both

1. *behavioral code* running on a PSE; with the notion of time and structures that forms part of the platform simulation environment, and
2. *functional code* running natively; that is closely linked to the underlying host system and may use host resources such as physical devices, for example USB port, Ethernet NIC, or software libraries, for example graphics.

A dynamic library (Linux *shared object* or Windows *dynamic link library*) is created to be loaded into the simulator that is bound to a peripheral instance. The dynamic library API binds functions in the peripheral (by name) to callback functions in the dynamic library.

To tell the simulator to load a dynamic library, set the extension field in the peripheral modelAttrs structure to the name of library (without it’s file extension) and put the library in the same directory as the peripheral model executable.

Dummy functions are created in the peripheral model and called when external functionality is required.

When the peripheral executes, calls to the bound functions are intercepted by the simulator causing the calling of the substitute functions in the dynamic library. Since the dynamic library can be linked with other host libraries (e.g. graphics or networking) the peripheral has access to any functionality that the host computer can support.

7.2 Constructor

7.2.1 Environment Checks

An important check to carry out as part of the peripheral model in the intercept portion of the code is to determine that the processor[4] type onto which the plugin has been loaded is a PSE.

```
    // What am I ?
    const char *procType = vmirtProcessorType(processor);
    if (strcmp(procType, "pse") != 0) {
        vmiMessage("F", PREFIX, "Processor must be a PSE\n");
    }
```

7.2.2 The Peripheral Simulation Engine ABI

The peripheral models are created using C code and APIs (BHM and PPM) on a PSE. A PSE is actually a processor model based on the host processor (in this case x86). This is used to provide an efficient (same instruction set used for PSE and Host) and yet safe (PSE cannot access outside its own memory) environment for code execution.

As with standard processor models parameters passed to intercepted functions and returned from intercepted functions do so using a fixed ABI. In the case of the PSE the ABI conforms to an x86 ABI for the provided toolchain.

In a typical constructor the registers can be determined and stored away for future use

```
    // return register (standard ABI)
    object->result = vmiosGetRegDesc(processor, "eax");

    // stack pointer (standard ABI)
    object->sp = vmiosGetRegDesc(processor, "esp");
```

7.3 Reading Intercepted Function Arguments

Arguments are passed on the stack. To read arguments the Stack Pointer (SP) must first be read. From the constructor the register containing the SP has already been determined and stored in the passed object.

The following is a standard function to read argument *index* from the intercepted function.

```
//
// Read a function argument using the PSE ABI
//
static void getArg(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         index,
    void         *result
) {
    memDomainP domain    = vmirtGetProcessorDataDomain(processor);
```

[4] For modeling and simulation purposes both processors and peripherals are thought of as processors.

```
    Uns32      argSize   = 4;
    Uns32      argOffset = (index+1)*argSize;
    Uns32      spAddr;

    // get the stack
    vmiosRegRead(processor, object->sp, &spAddr);

    // read argument value
    vmirtReadNByteDomain(domain, spAddr+argOffset, result, argSize, 0, True);
}
```

7.4 Passing the Return Code from an Intercepted Function

The function result is passed in a register. The register in which the result is passed is determined in the constructor and stored in the passed object. The result is written into this register using the *vmiosRegWrite* function.

```
   Bool result = nativeFunction();

   vmiosRegWrite(processor, object->result, &result);
```

7.5 Efficient Data Exchange

The use of the intercept library provides an efficient method for passing data between the simulation and the host environments.

The following code snippet shows how data can be transferred from the protected PSE memory domain into the native host domain.

Initially arguments have been read from the intercepted functions stack to determine the address in PSE memory space and the size of the buffer to transfer, stored in the local variables *vgaRamAddr* and *vgaRamSize* respectively. The data can then be transferred from the PSE memory domain into a native host memory buffer, *ramCopy*.

```
memDomainP domain = vmirtGetProcessorDataDomain(processor);

// copy ram into local address space
vmirtReadNByteDomain(domain, vgaRamAddr, ramCopy, vgaRamSize, 0, True);
```

7.6 Example Peripheral

In this example the function in the PSE *periphDisplay* will be performed by the native function *hostDisplay* that is able to call into a native function to create a graphics window on the host machine.

Within the PSE behavioral model a stub function is defined with a call to a "Fatal" message. This function should always be intercepted and so this stub should never execute.

```
#include "bhm.h"

//define no inline attribute for intercepted functions
#define NOINLINE __attribute__((noinline))

static Uns8 graphicsRam[64];

NOINLINE void periphDisplay(Uns8 *ram, Uns32 size){
    // fatal error to reach this point
    bhmMessage("F", "MY_MODEL", "This function should be intercepted");
}

int main(){
    // all coded in the main thread, though need not be.
    while(1) {
        bhmWaitDelay(frameUpdatePeriod);
        periphDisplay(graphicsRam, sizeof(graphicsRam));
    }
}
```

- The intercepted routine must be declared so that it is not inlined as these functions cannot be intercepted

The native code of the dynamic library contains the interface to the PSE to allow argument and result passing and data exchange.

```
// VMI includes
#include "vmi/vmiMessage.h"
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiOSLib.h"
#include "vmi/vmiRt.h"
#include "vmi/vmiTypes.h"
#include "vmi/vmiVersion.h"

typedef struct vmiosObjectS  {

    // return register (standard ABI)
    vmiRegInfoCP result;

    // stack pointer (standard ABI)
    vmiRegInfoCP sp;

} vmiosObject;

//
// Read a function argument using the PSE ABI
//
```

```
static void getArg(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         index,
    void         *result
) {
    memDomainP domain    = vmirtGetProcessorDataDomain(processor);
    Uns32      argSize   = 4;
    Uns32      argOffset = (index+1)*argSize;
    Uns32      spAddr;

    // get the stack
    vmiosRegRead(processor, object->sp, &spAddr);

    // read argument value
    vmirtReadNByteDomain(domain, spAddr+argOffset, result, argSize, 0, True);
}

static VMIOS_INTERCEPT_FN(hostDisplay)
{
    Uns32 vgaRamSize;
    Uns32 vgaRamAddr;
    getArg(processor, object, 0, &vgaRamAddr);
    getArg(processor, object, 1, &vgaRamSize);

    // call graphics engine.
    Uns8 ramCopy[vgaRamSize];

    memDomainP domain = vmirtGetProcessorDataDomain(processor);

    // copy ram into local address space
    vmirtReadNByteDomain(domain, vgaRamAddr, ramCopy, vgaRamSize, 0, True);

    // do the work
    doGraphics(ramCopy, vgaRamSize);
}


static VMIOS_CONSTRUCTOR_FN(constructor)
{
    vmiMessage("I" ,"VGA_UP", "Starting");

    // return register (standard ABI)
    object->result = vmiosGetRegDesc(processor, "eax");

    // stack pointer (standard ABI)
    object->sp = vmiosGetRegDesc(processor, "esp");
}

vmiosAttr pluginAttrs = {
    VMI_VERSION,                // version string (THIS MUST BE FIRST)
    "vga",                      // description
    sizeof(vmiosObject),        // size in bytes of object

    constructor,                // object constructor
    0,                          // object destructor
    0,                          // morph callback
    0,                          // disassemble instruction

    // ADDRESS INTERCEPT DEFINITIONS
    // -------------------   ----------- ------ -----------------
    // Name                 Address     Opaque Callback
    // -------------------   ----------- ------ -----------------
```

```
    {
      {"periphDisplay",      0,          True,  hostDisplay },
      { 0 }
    }
};
```

8 Creating a Simple DMA Controller

This section provides an example that introduces some of the peripheral modeling aspects introduced in the earlier sections.

8.1 Example System

The peripheral example described over the next sections is a DMA controller in a system with a single OR1K processor and two memory regions. This is a simplistic DMA model to introduce the main aspects of peripheral development and does not represent a particular DMA device.

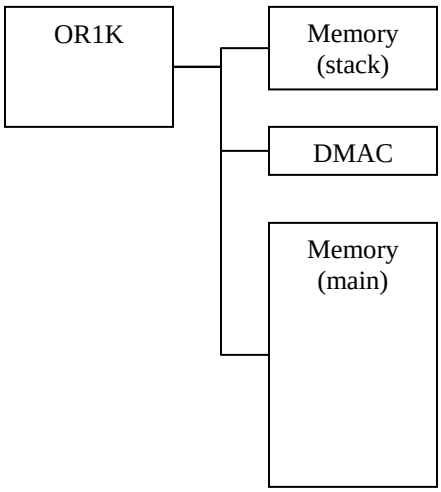


Figure 1: Example Virtual Platform Block Diagram

8.2 Example Source

The source files for this example can be found in the OVP / Imperas installation in the directory:

```
Examples/Models/Peripherals/creatingDMAC
```

This document introduces new material in an incremental way. To see the full source code of the peripheral model the example files should be inspected while reading the document. The file `README.txt` in the `creatingDMAC` directory describes the files which make up the example and how to run the example.

The example files accompanying this document are structured as a progression of self-contained implementations, each step building incrementally on the last.

8.2.1 Peripheral Registers

(example directory `1.registers`)

The initial example introduces the peripheral structure and the instantiation of the peripheral in a virtual platform harness. This example allows a master device that resides on the same bus as the peripheral access to registers within the peripheral. The peripheral registers are defined and (empty) callback functions installed to implement their behavior.

8.2.2 Parallel Operations and Signaling Events

(example directory `2.parallelThreadsAndEvents`)

The previous example is extended to add behavior to the peripheral registers when there are accesses to the peripheral registers. Two threads are added to model the two DMA channels which can run simultaneously.

Note that the user-data field which is this example is used as a pointer to the current register value, so that a single callback function can be re-used for several registers.

8.2.3 Master Memory Access

(example directory `3.memAccess`)

This example shows the creation of bus master ports on the peripheral that can be used to perform memory access and transfers from memory, or other memory mapped devices in the virtual platform. The DMA burst function is completed. In this contrived example, the DMA channel performs short bursts of reads and writes followed by a short wait to simulate the time taken to perform the DMA.

8.2.4 Interrupts

(example directory `4.interrupt`)

This example shows the addition of output interrupt nets to the DMAC controller so that it can signal an event, such as end of DMA transfer, to another device in the platform (the processor in this case). The interrupts are connected between devices using nets. A reset input net is added (which if unused, does not need to be tied inactive).

8.2.5 Behavior using Native host code

(example directory `5.nativeBehaviour`)

Finally, this example illustrates the use of native host code to provide behavior in a peripheral model. One of the two DMA channels is configured so that native code directly transfers data into the simulation memory. In this case we are using native code to copy between the DMA source and destination. It is more likely, that native code is being used so that the data can be efficiently passed to another host program or device.

8.3 Simulation Platform Overview

The virtual platform is created using a standard C file and the OVP ICM API functions. This allows the instantiation of, for example, processors, memories and peripheral devices and their connection using buses and nets[5].

The Open Virtual Platform initiative provides a simulation engine and example processor and peripheral models that are loaded into the virtual platform.

This example refers to an opencores OR1K processor model and generic memory model.

For a full description of virtual platform creation commands see the "CpuManager and OVPsim User Guide".

8.3.1 Virtual Platform Design

This section is intended as a brief summary of the virtual platform creation.

8.3.1.1 Virtual Platform Memory Map

Figure 2 shows the memory map of the virtual platform.

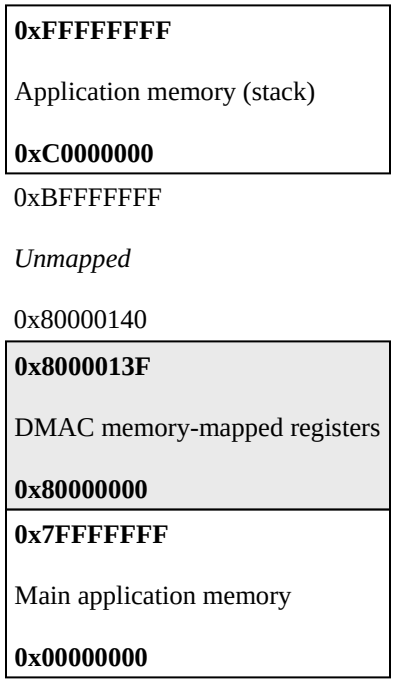


Figure 2: Memory map of platform

8.3.1.2 OVP Virtual Platform

The following provides the OVP definition of the platform.

[5] Nets are used, amongst other things to create asynchronous interrupt lines between components.

It comprises, the OVP environment initialization

```
icmInitPlatform(ICM_VERSION, ICM_VERBOSE|ICM_STOP_ON_CTRLC, 0, 0, “platform”);
```

The creation of a main bus onto which the devices can be connected

```
 icmBusP bus = icmNewBus("busMain", 32);
```

The creation of memory devices and their connection to the bus. The base addresses of the memories are defined when they are connected to the bus and the size of the memories is defined when they are created.

```
 // create two memory regions
 icmMemoryP mem1 = icmNewMemory("mem1", ICM_PRIV_RWX, 0x3fffffff);
 icmMemoryP mem2 = icmNewMemory("mem2", ICM_PRIV_RWX, 0x7fffffff);

 // connect memories to the bus
 icmConnectMemoryToBus(bus, "sp", mem1, 0xc0000000);
 icmConnectMemoryToBus(bus, "sp", mem2, 0x00000000);
```

The processor is instantiated. Notice that the information to define the processor is passed from the Makefile during compilation.

```
// create a processor instance
icmProcessorP processor = icmNewProcessor(
  "CPU1",             // CPU name
  CPU_TYPE,           // CPU type
  0,                  // CPU cpuId
  0,                  // CPU model flags
  32,                 // address bits
  MORPHER_FILE,       // model file
  MORPHER_SYMBOL,     // morpher attributes
  SIM_ATTRS,          // attributes
  0,                  // user-defined attributes
  SEMIHOST_FILE,      // semi-hosting file
  SEMIHOST_SYMBOL     // semi-hosting attributes
);
```

The processor has both its instruction and data ports connected to the bus.

```
// connect the processor instruction and data busses to the bus
icmConnectProcessorBusses(processor, bus, bus);
```

The peripheral is instantiated in the platform and connected to the bus. The port name ‘DMACSP’ of the slave port must match the name in the peripheral model as must the size of the port that is exposed during the call to the ppmOpenSlaveBusPort.

```
// instantiate the peripheral
icmPseP dmac = icmNewPSE("dmac", "dmacModel.pse", NULL);

// connect the DMAC slave port on the bus
//define the address range it occupies
icmConnectPSEBus(dmac, bus, "DMACSP", False, 0x80000000, 0x8000013f);
```

The platform uses the Imperas Command Line Parser to give access to standard capabilities such as loading the program file using the '--program' argument.

The simulation is started using the call to *icmSimulatePlatform* and will run until it is either interrupted; this was allowed when the simulation environment was initialized; or it runs to exit.

```
// simulate the platform
icmProcessorP final = icmSimulatePlatform();

// was simulation interrupted or did it complete
if(final && (icmGetStopReason(final)==ICM_SR_INTERRUPT)) {
  icmPrintf("*** simulation interrupted\n");
}
```

This file is compiled as a standard native executable using the provided Makefile.

8.4 Creating a Peripheral Model

Peripheral code is written in C and compiled using a compiler toolchain to run on a dedicated processing engine known as the Peripheral Simulation Engine (PSE). The code uses the BHM and PPM libraries to interact with the simulator and produce the behavior of the peripheral.

8.4.1 Peripheral Model Code Entry

The peripheral model is created as a standard C file, including the peripheral modeling header files. As such, it contains a main() function that is the entry point when simulation starts.

```
//
// Main for  DMAC
//
int main(int argc, char **argv) {
    bhmInstallDiagCB(setDiagLevel);
    busPortConnections();
    netPortConnections();
    userInit();
    return 0;
}
```

In the example the main function first installs a local function used to change the diagnostic level. This function should set a local variable which is then tested to control diagnostic output.

```
//
// local function to change diagnostic level
//
static void setDiagLevel(Uns32 newValue) {
  globalDiagFlag = newValue;
}
```

Main then calls several initialization functions to add port and net connections and perform initialization such as register initialization before calling the 'userMainLoop'. In fact in the first example only the bus port connection is made in the BusPortConnections function, the other functions contain diagnostic messages only.

8.5 Accessing the Peripheral

The peripheral can be a slave device that is able to connect to a bus and expose register and/or memory to be accessed from that bus. This would provide a memory mapped slave interface. Accesses by a bus master to this slave exposed region control the peripheral's behavior and return information about the peripheral's state. Typically this is in the form of memory mapped registers for passing control and returning status information but can also be a memory region.

8.5.1 Data Ordering

It is important to note that the PSE executes as a little-endian device and, if required, the interface between the platform bus and the peripheral should perform the conversion. In this example we have a small function to perform the conversion[6]. This ensures that the correct data is provided at the peripheral interface for little-endian or big-endian platforms.

[6] This is enabled by a parameter in the Makefile when compiled. It can also read the endian from the platform.

```
  data = byteSwap(data);
```

8.5.2 Creating a Slave Port Interface

The peripheral slave port occupies a memory window in the system address space. The size of the memory window is defined as part of the peripheral; the position is defined as part of the design in which the peripheral is used.

The window requires that there is an area of storage defined within the peripheral. This area is the source and destination of all slave accesses to the peripheral.

```
  static unsigned char DMACSP_Window [0x140];
```

The memory window into the peripheral is exposed to the rest of the system by opening a slave bus port using the PPM command **ppmOpenSlaveBusPort**.

```
    DMACSP_handle = ppmOpenSlaveBusPort(
        "DMACSP",
        DMACSP_Window,
        sizeof(DMACSP_Window)
    );
```

The port, ***DMACSP***, identifies the bus slave port within the design and provides the link between the peripheral software and the platform. The port provides access to the memory region, ***DMACSP_Window***, in the peripheral that is ***sizeof(DMACSP_Window)*** bytes in extent.
The port does not have to be opened across the whole extent of the local memory; it can be a subset. There may be any number of slave ports created on the peripheral that can access part or all of the same local memory region.

This now allows a master device to read and write to the exposed area of memory within the peripheral device. At this stage the peripheral does not perform any useful operations.

8.5.3 Register Behavior

A write to the exposed peripheral window by a bus master will result in the data value being stored in the memory array. A read from the window will return the value in the exposed memory array which corresponds to the address accessed.

When a read or write access in the memory window is to an address that requires functionality in addition to the memory access, for example, a write to a control register, callback functions are installed to implement that functionality.

There are two methods to trigger code in the PSE from a read or write to its slave port(s);

- **ppmCreateRegister** attach read and/or write callbacks, but also define a register object which is visible to the debugger as part of the platform user view. This method is preferred in most cases.
- **ppmInstallReadCallback** and **ppmInstallWriteCallback** attach callbacks to a specific memory region. This is a manual method that does not provide visibility to the debugger and Imperas analysis tools.

8.5.3.1 Installing a Register

When creating a register in a peripheral model the recommended approach is to use the *ppmCreateRegister* or the *ppmCreateInternalRegister* API functions.

Both functions create an instance of a register in the peripheral model. The register, defined in this manner, has a name and description but also can be provided with a function used by the debugger to allow access without side effects. Also as part of this API an 'event' is created that can be reported whenever the register is read or written.

Figure 3 shows two register types within an example peripheral model. One 'config' is accessible through a port and the other 'runstate' is an internal register, only accessible from within the model itself. The following paragraphs show how these are created using the API functions.

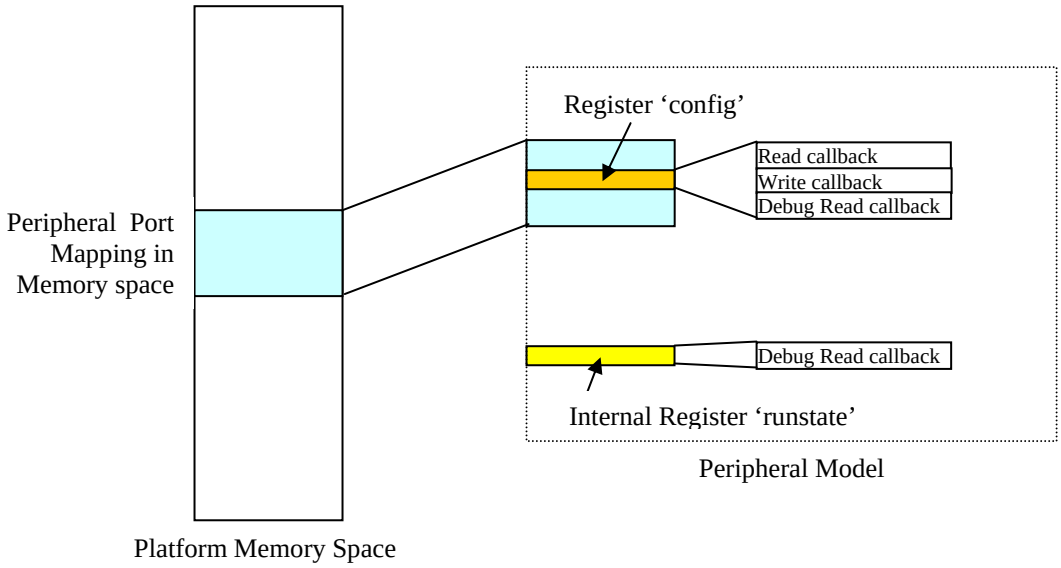


Figure 3: Example Peripheral Registers

The *ppmCreateRegister* combines the association of read and write callbacks, that provide the behavior behind a register access, with an offset within the memory representation behind the port.

```
ppmCreateRegister(
    "config",                               // name
```

```
        "configuration",                        // description
        DMACSP_Window,                          // window base
        0x30,                                   // port offset in bytes
        1,                                      // register size in bytes
        readRg8,                                // read callback
        configWr,                               // write callback
        viewReg8,                               // debug view
        &control.config                         // user data
    );
```

In the above example a register described as *configuration* that is named *config* is created in the peripheral. This register is accessible through the port associated with the memory region `DMACSP_Window` at an offset of 0x30 byte from the base. The behavior of accessing the register is defined in the two callbacks `readRg8` and `configWr` for read and write access respectively. An element from a control structure (control.config) holds the state of the register and a pointer to this is passed in the user data argument. A separate function is provided to give the debugger read access of the register without any side effects, that may be modeled by accessing through the normal read callback function.

The *ppmCreateInternalRegister* does not have any association with a port and is internal to the peripheral.

```
    ppmCreateInternalRegister(
        "runstate",                             // name
        "operational status",                   // description
        1,                                      // register size in bytes
        viewReg8,                               // debug view
        &runstate                               // user data
    );
```

In the above example an internal register described as *operational state* and named *runstate* is created within the peripheral. This register is not accessible through a port of the peripheral. The value is stored in the *runstate* variable, a pointer to which is passed.

8.5.3.2 Installing Callbacks

Any number of callbacks may be associated with a port. This is very useful when the peripheral, like the DMAC example, has a number of separate register groups. In the DMAC example these are the general control/common registers and the specific channel control registers.

Callback functions are installed separately for read and write accesses using the **ppmInstallReadCallback** and **ppmInstallWriteCallback** functions respectively, as shown for the installation of the channel 1 register callbacks. Callbacks should always be declared using the provided macros, to minimize the effects of API changes.

```
ppmInstallReadCallback(
      DMACSP_channel1Registers_Read_portCB,
      0,
      &DMACSP_Window[0x120],
```

```
        0x14
    );

    ppmInstallWriteCallback(
        DMACSP_channel1Registers_Write_portCB,
        0,
        &DMACSP_Window[0x120],
        0x14
    );
```

The callback function ***DMACSP_channel1Registers_Read_portCB***: is the name of the read callback function to be invoked when there is an access in the memory window. The callback is associated with a range of the exposed window defined by array ***DMACSP_Window[]***. This callback is active over a range within this window, starting at the address specified by ***&DMACSP_Window[0x120]*** and covering ***0x14*** bytes.
There is no user data passed to the callback function when it is installed so ***0*** is passed as the user data parameter.

The memory window may be common for both the read and write callbacks.

```
PPM_READ_CB(DMACSP_controlRegisters_Read_portCB){
    if(bytes != 4) {
        addressError(addr, bytes, 4);
        return ;
    }

 … etc …
}

PPM_WRITE_CB(DMACSP_controlRegisters_Write_portCB){
    if(bytes != 4) {
        addressError(addr, bytes, 4);
        return ;
    }

 … etc …
}
```

The parameter ***addr*** is the address of the access in the PSE address space (it will always fall within the `DMACSP_Window in this example)`, ***bytes*** is the size of the access in bytes, ***user*** is user specific data that was passed to the install function and ***data*** is taken from the bus to be used in the callback function.

Often a peripherals registers are all accessed as the same size, or only certain access sizes are supported. To ensure no incorrect behavior, a check of the access size and the reporting of a violation has been added to the model.

Also added to the model is the byte swapping of the word data as it is received in the write functions or returned from the reading functions.

The ***addr*** parameter can be converted to an offset in the slave port by subtracting the

window base address:

```
Uns32 offset = (unsigned char*)addr - &DMACSP_Window[0x0];
```

The 4th argument to the callback is a boolean which is true when the access is a simulation artifact. This occurs when the simulator is pre-fetching values for dynamic code translation, or originates from a debugger. The model might decide to inhibit side effects for this kind of access.

8.5.4 Simulating a bus error in a callback

In a register or memory callback it is possible to abort the read or write access that is currently in progress on the application processor, or on another peripheral acting as a bus master, depending on how this peripheral was activated. To do this, call:
`ppmReadAbort()`
or
`ppmWriteAbort()`

In the case that the peripheral was activated by an application processor, and simulated exceptions are enabled, the processor's read or write abort exception handler functions will be called. Typically, these will cause the processor to jump to an exception vector to handle the abort. If simulated exceptions are not enabled, the simulator will stop, reporting that an unhandled processor exception has occurred.

To abort another peripheral acting as a bus master, the bus master peripheral must use functions
`ppmReadAddressSpace()`
or
`ppmWriteAddressSpace()`
to initiate a bus transaction. If this bus transaction is aborted by another peripheral, the functions return False, and the address that caused the abort can be found by calling
`ppmGetAbortAddress()`

8.5.5 Adding the Peripheral to the Platform

The peripheral is added as a PSE into the virtual platform. Note that the port name and port size must match the port created with the ***ppmOpenSlaveBusPort*** function.
If there is a discrepancy it can lead to indeterminate behavior and run-time failures.

```
// instantiate the peripheral
icmPseP dmac = icmNewPSE("dmac", "dmacModel.pse", NULL);

// connect the DMAC slave port on the bus
// define the address range it occupies
icmConnectPSEBus(dmac, bus, "DMACSP", False,
                 0x80000000, 0x8000013f);
```

When the peripheral is instantiated it is given a name, the name of the compiled executable that describes its behavior and an attribute list. There are no attributes passed in this example but these can be used to configure the peripheral behavior.

8.5.6 Running the Example

This section has described the initial stage of peripheral development covered in the example ***1.registers***. The following provides the commands to run the example and illustrates the expected output

In the directory ***Examples/Models/Peripherals/creatingDMAC/1.registers*** there is a Makefile that will build the virtual platform, the test application software and the peripheral PSE behavioral code.
In the directory use the command:

```
> make clean all
```

The platform simulation is performed with the following command:

```
> ./platform/platform.${IMPERAS_ARCH}.exe --program application/dmaTest.elf
```

Output similar to the following should be observed:

```
OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  started: Tue Mar 17  12:58:47 2008


Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/processor/or1k/1.0/model.dll'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model
.dll'
TEST DMA: dmaBurst ch:0  bytes:13
TEST DMA: dmaBurst ch:1  bytes:35
TEST DMA: DMAC Register Read 00000000
TEST DMA: DMAC Register Read 00000000
TEST DMA: DMAC Register Read 00000000
TEST DMA: DMAC Register Read 00000000
TEST DMA: DMAC Register Read 00000000
TEST DMA: DMAC Register Read 00000000
TEST DMA: DMAC Register Read 00000000
TEST DMA: DMAC Register Read 00000000
Info
Info ------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    0.01 seconds: PSE 'dmac' (and 17 terminated threads)
```

```
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info CPU 'CPU1' STATISTICS
Info   Type                  : OR1K
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1f38
Info   Simulated instructions: 122,112
Info   Simulated MIPS        : 3.4
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.05 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.04 seconds
Info ---------------------------------------------------


OVPsim  finished: Tue Mar 17  12:58:47 2015
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
```

8.6 Simulating Multiple Parallel Operations within a Peripheral

A peripheral may contain a number of processes that operate in parallel, either independently or interactively. In order to model this behavior each process can be represented by a separate software thread. Interactions between parallel processes are modeled by events.
The example code in ***2.parallelThreadsAndEvents*** will add further functionality to the code from example code in ***1.registers*** to support two DMA channels operating in parallel and controlled by events.

8.6.1 Creating New Peripheral Threads

An initial thread is created by the main() routine within the peripheral; any additional threads that are required within the peripheral are created using the **bhmCreateThread** function.

An area to be used as a stack must be defined for every additional thread. In the example a character array is defined in the structure for each of the DMA channels control and state; this also holds a handle to the thread when created and also a handle to an event.

```
#define THREAD_STACK    (8*1024)

typedef struct {
    bhmThreadHandle       thread;
    bhmEventHandle        start;
    Bool                  busy;
```

```
    char                  stack[THREAD_STACK];
} channelState;
```

Each of the channels for the DMA controller are created in the same way. An event is defined in the structure that will allow the channel to be 'started' and the thread is created.

```
// Create threads for the channels
for (i=0; i<NUM_CHANNELS; i++) {
   // Event to start the thread

   DMAState.ch[i].start = bhmCreateEvent();
   DMAState.ch[i].busy = False;

   sprintf(threadName, "ch%u", i);
   DMAState.ch[i].thread = bhmCreateThread(
               channelThread,
               (void*) i,
               threadName,
               &DMAState.ch[i].stack[THREAD_STACK]     // top of downward
                                                       growing stack
         );
}
```

When the thread is created it is provided the name of the function invoked for the new thread, ***channelThread***, the channel number, used to configure the common function that is used for the DMA channel behavior, is being passed in the user data as a void pointer, ***(void*) i***, and a pointer, ***&DMAState.ch[i].stack[THREAD_STACK]***, to the top of the region to be used for the thread stack.

8.6.2 Using Events to Synchronize and Control Threads

A thread will free run in relation to another unless a synchronization method is used. One such synchronization method is provided in the OVP environment using events. Once an event is created threads can cause an event or wait on an event.

8.6.2.1 Creating Events between Threads

We saw in the previous section an event being created and the returned handle being stored in the channel structure. This handle should be stored such that it is accessible to any thread wishing to operate on the event.

```
// Event to start the thread
DMAState.ch[i].start = bhmCreateEvent();
```

The event handle ***DMAState.ch[i].start*** is used in the functions to wait or trigger an event.

8.6.2.2 Waiting for an Event

A thread can be caused to stop until an event occurs using the function **bhmWaitEvent()**. The thread will occupy no simulation time while waiting for an event.

In the DMA channel function, ***channelThread***, the behavior of the channel is created. The channel will perform some operations and then will wait until the start event, ***DMAState.ch[ch].start***, is triggered.

```
static void channelThread(void *user)
{
    Uns32 ch = (Uns32) user;
    for (;;) {

        … code …

        bhmWaitEvent(DMAState.ch[ch].start);

        … code …
    }
}
```

In this example we are not making use of the return value from the bhmWaitEvent function.
The return value indicates if the event triggered, was deleted or if the event handle was invalid.

8.6.2.3 Triggering an Event

An event can be triggered using the function **bhmTriggerEvent()**. In this example the start event for a channel is caused by the behavioral code of a write to the configuration register of that channel. If the appropriate bits in the configuration register are set the event is triggered.

```
void  DMACSP_channel0Registers_configuration_Write_userCB ...

   … code …

   if(!DMACSP_channel0Registers_I.configuration.bits.halt &&
       DMACSP_channel0Registers_I.configuration.bits.enable &&
      !DMAState.ch[0].busy) {

           bhmTriggerEvent(DMAState.ch[0].start);
    }
}
```

In this example the return value of the bhmTriggerEvent function is not used. If used it returns true when the event handle ***start*** exists.

8.6.2.4 Deleting an Event

An event can be deleted. If an event is deleted any threads waiting on the event are restarted and will no longer stop on that event.

```
bhmDeleteEvent(DMAState.ch[0].start);
```

The bhmDeleteEvent function returns true when the event handle ***start*** exists and the event is deleted.

8.6.3 Add a Delay to a Thread

It may be required to add some time aspects to a function. A delay can be achieved using the function **bhmWaitDelay**. The delay is specified in microseconds as a double.

In this example a delay is introduced between the DMA burst completing and the status bit indicating it has completed being set.

```
... etc ...
 // Perform DMA burst
 dmaBurst(ch);

 bhmWaitDelay(10); // wait 10 microseconds

 DMACSP_controlRegisters_I.rawIntTCStatus.value |= (1 << ch);

... etc ...
```

The time specified for the delay is simulated time. The delay is based upon a microsecond resolution but a double is used so a fraction may be passed.

8.6.4 Running the Example

This section has described the addition of parallel DMA channels that are enabled from the access to the DMA controller registers covered in the example ***2.parallelThreadsAndEvents***. The following provides the commands to run the example and illustrates the expected output.

In the directory
Examples/Models/Peripherals/creatingDMAC/ 2.parallelThreadsAndEvents
there is a Makefile that will build the virtual platform, the test application software and the peripheral PSE behavioral code.
In the directory use the command:

```
> make clean all
```

The platform simulation is performed with the following command:

```
> ./platform/platform.${IMPERAS_ARCH}.exe --program application/dmaTest.elf
```

Output similar to the following should be observed:

```
OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  started: Tue Mar 17  13:00:17 2015


Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/processor/or1k/1.0/model.dll'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model
.dll'
TEST DMA: dmaBurst ch:0  bytes:13
TEST DMA: dmaBurst ch:1  bytes:35
Info
Info ------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    0.00 seconds: PSE THREAD 'dmac'
Info    0.00 seconds: PSE THREAD 'dmac'
Info    0.02 seconds: PSE 'dmac' (and 11 terminated threads)
Info ------------------------------------------------
Info
Info ------------------------------------------------
Info CPU 'CPU1' STATISTICS
Info   Type                  : OR1K
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1eb8
Info   Simulated instructions: 106,235
Info   Simulated MIPS        : 3.0
Info ------------------------------------------------
Info
Info ------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.03 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.04 seconds
Info ------------------------------------------------


OVPsim  finished: Tue Mar 17  13:00:17 2015
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
```

The output now shows the two DMA channels, ch0 and ch1, implemented as peripheral threads performing burst transfers of 13 and 35 bytes respectively. The processor is polling the status registers of the two channels for confirmation that the DMA burst have completed. When complete the burst destination buffers are checked for correct data. It

should be noted that there is no confirmation that the data was actually transferred correctly to the destination. The peripheral, in this example, is not able to access the memory of the platform. The ports to allow master accesses into memory in the platform will be introduced in the following section.

8.7 Peripheral Bus Master Accessing Memory Regions

A peripheral device can be given the ability to access memory within the system as a bus master.
The example code in ***3.memAccess*** will add further functionality to the code from the example code in ***2.parallelThreadsAndEvents*** to allow master bus ports to be created that allow master accesses of the memory mapped in the virtual platform.

The access through a bus master interface by the peripheral model is implemented using PPM API functions. There are two forms of access, those that create an address space and those that open a port.

8.7.1 Opening an Address Space

An address space access is ideal when general access is required to a large memory area. Unlike a port access, see section 8.7.3, no local window area is required in the peripheral model and access is through accessor functions. The use of port access may be preferred for predictable access to a well-defined small address region port.

In the DMAC example the address space approach is used to allow access to the complete available memory space in the platform.
The structure used to hold the DMA state has had two additional fields added to hold handles of type ppmAddressSpaceHandle.

```
typedef struct {
    ppmAddressSpaceHandle readHandle;
    ppmAddressSpaceHandle writeHandle;
    channelState          ch[NUM_CHANNELS];
} dmaState;
```

The port is created using the function *ppmOpenAddressSpace*.

```
  DMAState.readHandle = ppmOpenAddressSpace("MREAD");
  if(!DMAState.readHandle) {
      bhmMessage("E", "DMAC", "Failed to open read port\n");
      bhmFinish();
  }
```

The field passed in the function, MREAD, is the name of the port that identifies the bus master port within the model definition and provides the link between the peripheral software and the platform.

8.7.2 Accessing an Address Space

Read and write functions allow any amount of data to be read from or written to the bus address space by the peripheral model. In the DMAC example the data is transferred in blocks of ***BYTES_PER_ACCESS*** bytes. The inner loop of the dmaBurst function is shown below:

```
while (bytes) {
    Uns32 thisAccess = (bytes > BYTES_PER_ACCESS) ?
                                        BYTES_PER_ACCESS : bytes;
      ppmReadAddressSpace (DMAState.readHandle,
                           src,  thisAccess, buff);
      ppmWriteAddressSpace(DMAState.writeHandle,
                           dest, thisAccess, buff);
 … etc …
```

The ***ppmReadAddressSpace*** and ***ppmWriteAddressSpace*** functions are used to respectively read ***thisAccess*** bytes of data from the physical address, ***src***, into the temporary local buffer specified by ***buff*** and write this data to the address, ***dest***.

8.7.3 Accesses using a Bus Master Port

Bus master port access works in much the same way as slave port access. A memory window is defined in the local memory area of the peripheral model which is then aliased to an address range on the bus. Once the port is open, accesses to the local area by the peripheral model implicitly cause accesses to the corresponding address on the bus. This form of access is useful when a peripheral has master access to memory-mapped registers of other devices.

8.7.3.1 Opening a Bus Master Port

The port is opened using a locally defined memory space and a platform memory address. Any access into the local memory space will provide a master access onto the bus based at the mapped address.

```
char localSpace[64];

ppmExternalBusHandle  busHandle;
busHandle = ppmOpenMasterBusPort("MPORT",
                                 &localSpace[0x0],
                                 sizeof(localSpace),
                                 0x10000000)
```

In this example the bus port name, ***MPORT***, is the name that identifies the bus master port within the model definition and provides the link between the peripheral software and the hardware design. The pointer, ***&localSpace[0x0]***, to the base of the local memory space in the peripheral and its size, ***sizeof(localSpace)***, are used to map accesses to the base address of the platform memory the bus port accesses.

8.7.4 Re-Locating the Window for Bus Master Port

Opening a master bus port presents a memory window through which an external address range can be accessed. This memory window can be moved within the platform memory space using the **ppmChangeRemoteLoAddress()** function.

```
ppmChangeRemoteLoAddress(busHandle,
                         0x10000100);
```

The handle, ***busHandle***, identifies the port that is re-located to the new base address. All future accesses into the local memory space will now cause a master access into the platform based at the new address.

8.7.5 Access through Bus Master Port

An access into the local memory space can now be used to cause an access into the platform memory.

This can perform a 32-bit integer access to a memory mapped register or memory location.

```
Uns32 register7;
Uns32 *p = localSpace;

register7 = p[7];
```

or byte accesses

```
Uns8 register1;
register1 = localSpace[1];
```

8.7.6 Adding the Master ports to the Platform

The bus master ports are added into the virtual platform. Note that the port name must match the port created with the ***ppmOpenAddressSpace*** function.
If there is a discrepancy it can lead to indeterminate behavior and run-time failures.

```
icmConnectPSEBus(dmac, bus, "MREAD",  True, 0x00000000, 0xffffffff);
icmConnectPSEBus(dmac, bus, "MWRITE", True, 0x00000000, 0xffffffff);
```

The PSE bus ports are created with master access set True and the peripheral is given access to the full extent of the 32-bit memory space.

8.7.7 Running the Example

This section has described the addition of master ports to allow the peripheral to perform bus master access covered in the example ***3.memAccess***. The following provides the commands to run the example and illustrates the expected output

In the directory ***Examples/Models/Peripherals/creatingDMAC/3.memAccess*** there is a Makefile that will build the virtual platform, the test application software and the peripheral PSE behavioral code.

In the directory use the command:

```
> make clean all
```

The platform simulation is performed with the following command:

```
> ./platform/platform.${IMPERAS_ARCH}.exe --program application/dmaTest.elf
```

Output similar to the following should be observed:

```
OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  started: Tue Mar 17  13:01:10 2015


Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/processor/or1k/1.0/model.dll'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model
.dll'
TEST DMA: dmaBurst ch:0  bytes:13
TEST DMA: dmaBurst ch:1  bytes:35
TEST DMA: DMA result Hello world. The whole world spread before you.
Info
Info ----------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    0.00 seconds: PSE THREAD 'dmac'
Info    0.00 seconds: PSE THREAD 'dmac'
Info    0.04 seconds: PSE 'dmac' (and 2,460 terminated threads)
Info ----------------------------------------------------
Info
Info ----------------------------------------------------
Info CPU 'CPU1' STATISTICS
Info   Type                  : OR1K
Info   Nominal MIPS          : 100
Info   Final program counter : 0x1e98
Info   Simulated instructions: 206,959
Info   Simulated MIPS        : 3.2
Info ----------------------------------------------------
Info
Info ----------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.08 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.06 seconds
Info ----------------------------------------------------


OVPsim  finished: Tue Mar 17  13:01:10 2008
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
```

In this example we see that the two channels are transferring 13 and 35 bytes of data respectively. The processor is polling the status registers of the two channels for confirmation that the DMA burst have completed. When complete the burst destination buffers are checked for correct data. Now we also see that the source data has been received correctly at the destination.

8.8 Using Interrupts and Signal Connections

Peripherals may be connected to other peripherals or processors with signal wires (nets). These can be used to act as interrupt signals or used to control behavior between peripherals.
The example code in ***4.interrupt*** will add further functionality to the code from the example code in ***3.memAccess*** so that the DMAC peripheral raises an interrupt when it has completed the data transfer. The interrupt line is created in the platform with a net, this is connected to a net port opened by the peripheral and another opened by the processor.

8.8.1 Opening a Net Port

A ***ppmNetHandle*** is added to the DMA control structure that will hold the handle to the net port when it is opened

```
typedef struct {
    ppmAddressSpaceHandle readHandle;
    ppmAddressSpaceHandle writeHandle;
    ppmNetHandle          intTCHandle;
    Bool                  intTCAsserted;
    channelState          ch[NUM_CHANNELS];
} dmaState;
```

The net port is opened and creates a port, INTTC, that is used to identify the net port within the platform.

```
  DMAState.intTCHandle = ppmOpenNetPort("INTTC");
  DMAState.intTCAsserted = False;
```

8.8.2 Generate an Interrupt

An interrupt is generated by writing a value to the net port handle using the ***ppmWriteNet*** function.

```
// Update TC interrupt line
if (DMAState.intTCHandle) {
    if (intTCStatus && !DMAState.intTCAsserted) {
        ppmWriteNet(DMAState.intTCHandle, 1);
        DMAState.intTCAsserted = True;
    } else if (!intTCStatus && DMAState.intTCAsserted) {
        ppmWriteNet(DMAState.intTCHandle, 0);
        DMAState.intTCAsserted = False;
    }
}
```

8.8.3 Reading Net Status

It is also possible for the peripheral to read the status of a net using the ***ppmReadNet*** function.

```
  Uns32 netState;
  netState = ppmReadNet(DMAState.intTCHandle);
```

8.8.4 Adding the Net ports to the Platform

The net ports are connected to the net in the virtual platform. Note that the port name must match the port created with the ***ppmOpenNetPort*** function.

A new net is created in the platform

```
    // declare a net for connection of the interrupt line
    icmNetP int1 = icmNewNet("int1");
```

this is connected as an input to the net port, ***intr0***, on the processor

```
    // connect the processor interrupt port to the net
    icmConnectProcessorNet(processor, int1, "intr0", ICM_INPUT);
```

and is connected as an output to the net port, ***INTTC***, on the peripheral

```
    // connect the DMAC interrupt port to the net
    icmConnectPSENet(dmac, int1, "INTTC", ICM_OUTPUT);
```

8.8.5 Running the Example

This section has described the addition of nets to implement interrupts covered in the example ***4.interrupts***. The following provides the commands to run the example and illustrates the expected output.

In the directory ***Examples/Models/Peripherals/creatingDMAC/4.interrupts*** there is a Makefile that will build the virtual platform and, the test application software and the peripheral PSE behavioral code.
In the directory use the command:

```
> make clean all
```

The platform simulation is performed with the following command:

```
> ./platform/platform.${IMPERAS_ARCH}.exe --program application/dmaTest.elf
```

Output similar to the following should be observed:

```


OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
Copyright (C) 2005-2015 Imperas Ltd.  Contains Imperas Proprietary Information.
Licensed Software, All Rights Reserved.
Visit www.imperas.com for multicore debug, verification and analysis solutions.
```

```
OVPsim  started: Tue Mar 17  13:02:16 2008


Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/processor/or1k/1.0/model.dll'
Info (ICM_AL) Found attribute symbol 'modelAttrs' in file
'C:\Imperas\lib\Windows\ImperasLib/ovpworld.org/semihosting/or1kNewlib/1.0/model
.dll'
TEST DMA: dmaBurst ch:0  bytes:13
TEST DMA: dmaBurst ch:1  bytes:35
TEST DMA: Waiting for interrupts
TEST DMA: Interrupt
TEST DMA: Interrupt ch0 0x8001
TEST DMA: Interrupt ch1 0x8001
TEST DMA: 2 interrupts received
TEST DMA: result Hello world. The whole world spread before you.
Info
Info ------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    0.00 seconds: PSE THREAD 'dmac'
Info    0.00 seconds: PSE THREAD 'dmac'
Info    0.03 seconds: PSE 'dmac' (and 15 terminated threads)
Info ------------------------------------------------
Info
Info ------------------------------------------------
Info CPU 'CPU1' STATISTICS
Info   Type                  : OR1K
Info   Nominal MIPS          : 100
Info   Final program counter : 0x20a0
Info   Simulated instructions: 213,039
Info   Simulated MIPS        : 4.5
Info ------------------------------------------------
Info
Info ------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 0.00 seconds
Info   User time             : 0.05 seconds
Info   System time           : 0.00 seconds
Info   Elapsed time          : 0.05 seconds
Info ------------------------------------------------


OVPsim  finished: Tue Mar 17  13:02:16 2008
Visit www.imperas.com for multicore debug, verification and analysis solutions.
OVPsim  v20150205.0 Open Virtual Platform simulator from www.OVPworld.org.
```

In this final example again the two channels transfer 13 and 35 bytes of data between the source and destination buffers. The processor is then waiting for an interrupt before reading the status registers of the two DMA channels. As we can see we receive an interrupt and poll the DMA status register. When both DMA bursts are complete the destination buffers are checked for correct data.

9 Building Peripherals

9.1 OVP Library Structure

It is recommended, as an example, the ImperasLib source tree structure is copied when creating peripheral models. There are example peripherals within this structure.

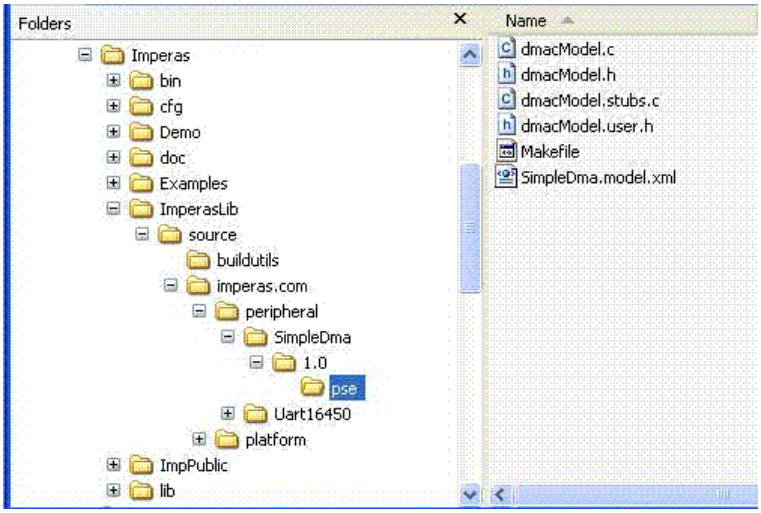


Figure 4: OVP Library Structure

This directory structure also incorporates a build system that allows the peripheral to be built. The build system makes use of Makefiles, all components in the ImperasLib/source directory can be re-built into either the default lib directory or into any specified directory.

9.2 Building Peripheral Models

In this section we discuss building a peripheral but this approach can build all component types found in a source library.

Using the peripheral simpleDma as an example for building, Makefiles are provided in

- $IMPERAS_HOME/ImperasLib/source, and
- $IMPERAS_HOME/ImperasLib/source/imperas.com/peripheral/SimpleDma/1.0/pse.
- $IMPERAS_HOME/ImperasLib/source/imperas.com/peripheral/SimpleDma/1.0/model[7].

These reference Makefiles in

- $ IMPERAS_HOME/ImperasLib/buildutils

that are used to build all the component types within the source library structure.

[7] This directory will only be present for models that build code to run natively on the host system. In a peripheral this would incorporate the intercept library.

9.2.1 Building to the Default Output Location

The default location for the output of the build is $IMPERAS_HOME/LIB/$ IMPERAS_ARCH

To build into the default directory, in an MSYS shell type

```
make –C $IMPERAS_HOME/ImperasLib/source
```

9.2.2 Building to a Defined Output Location

To build into a specified directory, for example a local directory ./myLocalLib, in an MSYS shell type

```
make –C $IMPERAS_HOME/ImperasLib/source VLNVROOT=$(pwd)/myLocalLib
```

10 Design Guidelines

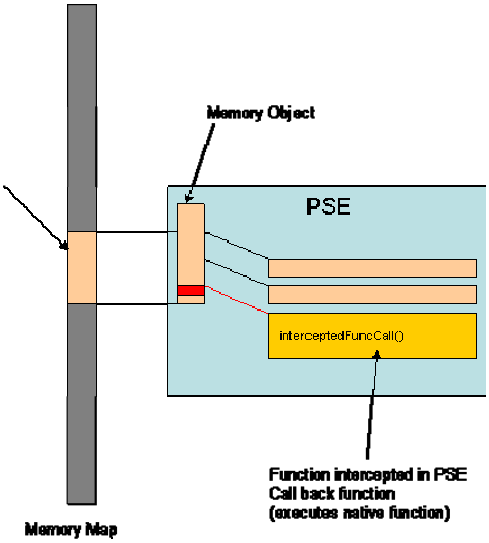
This section is intended to give some simple examples of common constructs when designing and building peripheral models.

10.1 Peripheral Registers

We are going to look at the peripheral model supplied as an example in Imperas/Model/Peripheral/creatingDMAC. This is a simple DMA Controller with two channels.

10.1.1 Creating Memory Mapped Registers

Typically registers are available to be accessed from other devices connected onto, or bridged onto, the same bus as the peripheral. This is modeled by exposing an area of memory, defined within the peripheral, by a port.
The peripheral exposes a port of a particular size but it is not until the peripheral is instantiated in a platform that this port is located in the platform memory space.




```
    static unsigned char DMACSP_Window[0x140];

...

static void createRegisters(void *w) {

    // common registers
    ppmCreateRegister("intStatus",      "internal status",  w, 0x00,  1,
                      regRd8, NULL,     viewReg8, &control.intStatus   );
...
    ppmCreateRegister("config",     "configuration",  w, 0x30,  1,
                      regRd8, configWr, viewReg8, &control.config.value );
...

    static unsigned char DMACSP_Window[0x140];
```

```
    DMACSP_handle = ppmOpenSlaveBusPort(
        "DMACSP",
        DMACSP_Window,
        sizeof(DMACSP_Window)
    );

    createRegisters(DMACSP_Window);
...
```

It is now possible for a device to access this memory area. This could be used as an internal memory area, for example scratchpad ram, within the peripheral or to maintain register values. If registers have no functionality i.e. they do not cause anything to happen when read or written the memory block is all that is required. However, if a read or write causes some action to be carried out within the peripheral we must add the mechanism to cause this behavior.

10.1.2 Call Back Function Gives Registers Behavior

A read and/or write call back can be applied to all or part of the memory block. When creating a register with *ppmCreateRegister* it is automatically applied to the area occupied by the register. When the memory covered by the call back function is accessed the function is called. Within the call back function the data is processed and any further behavior is described.

A callback could also be created separately from the *ppmCreateRegister* function using the `ppmInstallReadCallback`.

```
    ppmInstallReadCallback(
        DMACSP_controlRegisters_Read_portCB,    // call back function name
        0,                                      // user data
        &DMACSP_Window[0x0],                    // start address
        0x3c                                    // size
    );
```

Usually the function used as the call back function is associated with the memory mapped accesses to the peripheral through its exposed port and they would not be used for internal access to the registers within the peripheral.[8] The internal access to registers would bypass this function and access the register variable directly.

Note: When installing a callback across multiple possible register regions the address passed to the callback function is referenced to the actual address of the memory block defined within the peripheral model. The offset is calculated using the base address of the callback into this memory block.

```
PPM_WRITE_CB(DMACSP_channel1Registers_Write_portCB){
    if(bytes != 4) {
        addressError(addr, bytes, 4);
        return ;
```

[8] However, it is possible to also separately provide call backs that allow access from a native semihost part of the peripheral, see later in this document for an example.

```
    }

    data = byteSwap(data);
    Uns32 off = (unsigned char*)addr - &DMACSP_Window[0x120];
    switch(off) {
        case 0x0 :
            DMACSP_channel1Registers_srcAddr_Write_userCB(addr,bytes,data);
        break;
        case 0x4 :
            DMACSP_channel1Registers_destAddr_Write_userCB(addr,bytes,data);
        break;

… etc …

        break;
        default:
            DMACSP_channel1Registers_Default_Write_userCB(addr,bytes,data);
        break;
    }
}
```

In this example a separate function has been created to manipulate each register. The actual contents of the function may be as trivial as writing or reading a register variable.

```
PPM_WRITE_CB(DMACSP_channel1Registers_srcAddr_Write_userCB){
    if (BHM_DIAG_HIGH)
        bhmMessage("I", "PP_STUBS","Writing DMACSP/channel1Registers/srcAddr
(0x%x)\n", data);
    DMACSP_channel1Registers_I.srcAddr.value = data;
    // USER CODE DMACSP_channel1Registers_srcAddr_Write_userCB
}
```

10.1.3 Structure to Define Peripherals Registers

Normally, an underlying structure will be used to store the state of the peripheral. From within the peripheral these are accessed directly. From outside the peripheral they are accessed using the memory mapping of the registers and the applied callbacks.
Note: Defining this structure to hold the peripheral state allows us to easily create a union in the structure that can provide, for example, the register to be accessed both as a 32 bit integer or as bit fields.

```
typedef struct DMACSP_channel1Registers_StructS {
    union {
        Uns32 value;
    } srcAddr;
    union {
        Uns32 value;
    } destAddr;
    union {
        Uns32 value;
    } LLI;
    union {
        Uns32 value;
        struct {
            unsigned transferSize : 12;
        } bits;
    } control;
```

```
    union {
        Uns32 value;
        struct {
            unsigned enable :  1;
            unsigned __pad1 : 14;
            unsigned    itc :  1;
            unsigned _pad16 :  2;
            unsigned   halt :  1;
        } bits;
    } configuration;
} DMACSP_channel0Registers_Struct, * DMACSP_channel0Registers_StructP;
```

10.2 Using Native Host Code

10.2.1 When to Use

Normally a peripheral is defined completely within the PSE environment using the BHM and PPM API functions. This provides a safe environment ensuring accesses cannot be made into the rest of the simulated system that could cause corruption.

However, there may be specific functionality linked more closely with the native host, for example VGA display, keyboard input , USB or other peripheral devices. In these circumstances it may be more beneficial to create behavior that is modeled using native code. This cannot contain any of the timing provided by the BHM API calls, this must still be generated PSE behavior, but can provide functionality closely coupled to the host.

10.2.2 Function Interception

The method of communication between the two parts that now construct the peripheral (PSE and Native) is by function interception. The PSE is designed with functions and stubs that when called from within the PSE cause the simulator to intercept and run an alternative native function in their place. It is the native function that is used to provide the close coupling to the host system.

The stub function within the PSE contains only a “failure to intercept” warning message, as shown. This should be intercepted and never execute.

```
//
// Semihosted function: initializes input
//
NOINLINE void semiInit(InputStateP is)
{
    bhmMessage("F", PREFIX, "Failed to intercept semiInit");
}
```

The stub function must be specified with the attribute noinline to ensure that the symbol is visible.

```
#define NOINLINE __attribute__((noinline))
```

The native code makes use of the VMI API to transfer information from the simulation environment into the native environment. The functions in the PSE that are to be intercepted (and the functions that are run on interception) are defined as the last entry within the attribute table.

```
////////////////////////////////////////////////////////////////////
// INTERCEPT ATTRIBUTES
////////////////////////////////////////////////////////////////////

vmiosAttr modelAttrs = {

    ////////////////////////////////////////////////////////////////
    // VERSION
    ////////////////////////////////////////////////////////////////

    VMI_VERSION,                // version string (THIS MUST BE FIRST)
    "semiHostPeripheral",       // description
    sizeof(vmiosObject),        // size in bytes of object

    ////////////////////////////////////////////////////////////////
    // CONSTRUCTOR/DESTRUCTOR ROUTINES
    ////////////////////////////////////////////////////////////////

    constructor,                // object constructor
    destructor,                 // object destructor

    ////////////////////////////////////////////////////////////////
    // INSTRUCTION INTERCEPT ROUTINES
    ////////////////////////////////////////////////////////////////

    0,                          // morph callback
    0,                          // disassemble instruction

    ////////////////////////////////////////////////////////////////
    // ADDRESS INTERCEPT DEFINITIONS
    ////////////////////////////////////////////////////////////////
    // ------------------   ----------- ------ -----------------
    // Name                 Address     Opaque Callback
    // ------------------   ----------- ------ -----------------
    {
        {"semiInit",            0,       True,   inputInit },
        {"semiInputPoll",       0,       True,   inputPoll },
        {"semiReadData",        0,       True,   readData },
        {"semiWriteData",       0,       True,   writeData },
        {0}
    }
};
```

10.2.3 Reading Arguments Passed to a Function

A structure defined of type vmiosObject will be passed by the simulator into all the intercept functions defined using the VMIOS_INTERCEPT_FN macro. This is the preferred method of storing information about the registers the processor uses for passing arguments into functions and return values from functions.

```
// this structure holds information regarding the
```

```
// call and return registers used by the peripheral 'processor'
typedef struct vmiosObjectS  {

    // return register (standard ABI)
    vmiRegInfoCP result;

    // stack pointer (standard ABI)
    vmiRegInfoCP sp;

} vmiosObject;
```

The structure contains variables of type vmiRegInfoCP that are initialized to allow easy access to specific registers in the model. The VMI API function vmiosGetRegDesc is used in the constructor to initialize variables to the registers that hold the stack pointer and the function return value.

```
    // return register (standard ABI)
    object->result = vmiosGetRegDesc(processor, "eax");

    // stack pointer (standard ABI)
    object->sp = vmiosGetRegDesc(processor, "esp");
```

To obtain the arguments that have been passed to a function on the stack we must first read the register containing the stack pointer and then use this as the address to access in the processor address space.

The register is accessed using the vmiosRegRead function. This takes the register pointer that was initialized in the vmiosObject in the constructor.

To access the processor memory we must first obtain which of the memory domains[9] the processor is operating in. Using this information we use the vmirtReadNByteDomain function to read the correct number of bytes that contain the variable on the stack.

```
//
// Read a function argument using the standard ABI
//
static void getArg(
    vmiProcessorP processor,
    vmiosObjectP  object,
    Uns32         index,
    void         *result
) {
    memDomainP domain    = vmirtGetProcessorDataDomain(processor);
    Uns32      argSize   = 4;
    Uns32      argOffset = (index+1)*argSize;
    Uns32      spAddr;

    // get the stack
    vmiosRegRead(processor, object->sp, &spAddr);

    // read argument value
    vmirtReadNByteDomain(domain, spAddr+argOffset, result, argSize, 0, True);
}
```

[9] A memory domain describes an address space and related properties such as access permissions.

10.2.4 Data Structure/Buffer Passing

Using this getArgs function in an intercept routine, below, allows us to cleanly read a pointer off the stack and then use the same vmi function to use this pointer to read a structure into local memory.

```
//
// Called to start the service.
//
static VMIOS_INTERCEPT_FN(inputInit)
{
    Uns32 statep;
    InputState native;

    // Get the pointer off the peripheral stack
    getArg(processor, object, 0, &statep);

    // read data from the peripheral memory using the pointer
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtReadNByteDomain(domain, statep, &native, sizeof(native), 0, True);
```

10.2.5 Returning a Result

When a function provides a return value this could be passed back on the stack or directly in one of the processor registers. We can again use the information we initialized in the constructor to access the register used to pass the return value.

```
    // write return value into register
    vmiosRegWrite(processor, object->result, &result);
}
```

10.3 Configuring

10.3.1 Introduction

A peripheral model is configured using attributes. Attributes may be built into the platform description or passed on a command line invoking a simulation with the Imperas professional tools.

10.3.2 Defining in an ICM Platform

When a peripheral model has functionality created as both PSE and native code the attributes are all passed through the same attribute list to the instantiation of the PSE.

The following example code shows the creation of an attribute list to configure the peripheral model.

```
    // instantiate the VGA peripheral
  const char *vgaPse = icmGetVlnvString(vlnvRoot, "cirrus.ovpworld.org",
                              "peripheral", " GD5446", "1.0", "pse");
  const char *vgaIntercept = icmGetVlnvString(vlnvRoot, "cirrus.ovpworld.org",
                              "peripheral", "GD5446", "1.0", "intercept");
    {
  icmAttrListP vgaAttrs = icmNewAttrList();
```

```
  icmAddUns64Attr(vgaAttrs, "scanDelay", 50000);
  icmAddUns64Attr(vgaAttrs, "PCIslot", 18);
  icmAddUns64Attr(vgaAttrs, "noGraphics", noGraphics);
  icmAddStringAttr(vgaAttrs, "title", "OVPsim MIPS32 Malta");
```

The attribute list and the host functional code are both passed to the instantiation of the peripheral model.

```
  icmPseP vga = icmNewPSE("vga", vgaPse, vgaAttrs, vgaIntercept, "modelAttrs");
```

The final parameter on the instantiation allows the name of the attribute table specifying the access table for the simulator into the intercept component to be specified.

10.3.3 Defining on the Command Line

When the Imperas simulation environment is used a command line parameter may be used to set or override an attribute on a peripheral model.
For example, in the above ICM platform instantiation of the 'vga' peripheral model the 'noGraphics' attribute was used. This may be overridden using a command line such as:

```
  --override  MipsMalta/vga/noGraphics="1"
```

> ⇒ An attribute must exist in the platform in order for it to be overridden

10.3.4 Reading Configuration Attributes

BHM functions are used to access attribute lists that have been applied onto a peripheral model. The BHM functions allow access to different types of attributes, for example integer and string.

The following shows the reading of the 'noGraphics' integer attribute

```
  #define NO_GRAPHICS_ATTRIBUTE "noGraphics"
…
  Uns32 noGraphics = 0;
  bhmIntegerAttribute(NO_GRAPHICS_ATTRIBUTE, &noGraphics);
```

Using the BHM function shown above the variable *noGraphics* is only modified if the attribute with the specified string name is found in the attribute list.

11 Troubleshooting

This section is intended to give some examples of common problems, how they are typically caused and ways in which they can be overcome

11.1 Possible Runaway Recursion

11.1.1 Error Description

The peripheral modeling technology uses functions that are called back into from the simulator when there is an access to an address of an area of memory within a peripheral. This access could be from a processor memory mapped access to an exposed port of the peripheral or it could also be from an intercepted function that forms part of a peripheral.

The following illustrates the error as it would be seen when running a platform that contains a peripheral causing recursive calls.

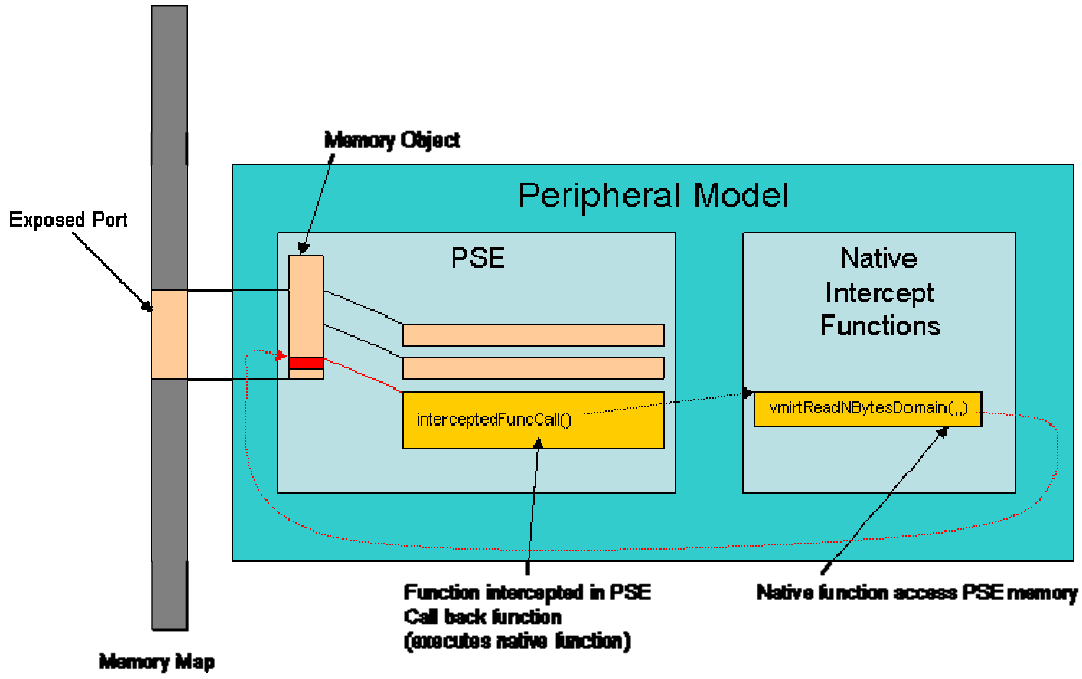
```
Error (PP_RR1) PSE basicPeripheral: Possible runaway recursion in read/write callbacks:
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 1 bytes at 0x80508a4
Fatal (PP_RR2) Cannot continue
Info Exiting
```

The simulator monitors the depth of callbacks occurring in the system and if it detects this is greater than a pre-defined maximum it terminate the simulation.

11.1.2 Example of Error in Peripheral Intercept Coding

The common problem is caused when addresses of regions of memory are passed from the PSE peripheral into the native peripheral intercept model from which they are used to access back into the PSE peripheral memory space.

The following diagram illustrates the problem of the native function using VMI API calls to access a memory region by address that resides in the PSE peripheral memory space.



The following is an example of code that could cause this recursion problem. The example is fairly simplistic but illustrates the problem.

A read port has been opened on the base of the memory window and a callback 'readPort' based at the address of window (which is a simple array) is created.

```
ppmInstallReadCallback(
    readPort,
    0,
    window,
    sizeof(window)
);
```

When there is an access to the memory addresses contained within the 'window' the call back 'readPort is called. This function contains a call to a dummy function (within the PSE) called semiReadData(). This is not actually ever executed as the simulator intercepts the function and instead calls a native function.

```
PPM_READ_CB(readPort){

    Uns32 val;

    semiReadData(&val);

    return val;

}
```

```
The native function semiReadData() uses a VMI API function to read from
an address
```

```
static VMIOS_INTERCEPT_FN(semiReadData)
{
    Uns32 count = 0;

    // Read data from the PSE data space
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    vmirtReadNByteDomain(domain, addrP, &count, sizeof(count), 0, True);

    vmiMessage("I", PREFIX, "Read Data: read %d from 0x%08x\n",
                             count, addrP);

}
```

The address has been passed from the PSE peripheral to the native intercept function during initialization.

```
void userInit(void)
{
    semiInit(window);
}
```

This is the address of the base of the window memory region in the PSE over which the read callback has been allocated. This read access is, therefore, to the same address that previously caused the original call back. It is going to cause the call back to be triggered once again and so cause the recursion.

##