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:	1.8.8
Filename:	OVP_Peripheral_Modeling_Guide.doc
Last Saved:	Friday, 26 February 2021
Keywords:	Peripheral PSE Modeling OVP

Table of Contents

1 Preface 7
1.1 Notation 7
1.2 Related OVP Documents 7
2 Introduction 8
2.1 Peripheral Model Template Generation with iGen 8
2.2 Prerequisites 8
2.2.1 MinGW 8
2.2.2 Cross Compiler Toolchain 8
3 OVP Simulation Overview 9
3.1 Imperas & OVP Tools 9
3.2 Processor Models 9
3.3 Peripheral Models 9
4 Behavioral Modeling Methodology 10
4.1 Peripheral Modelling API 10
4.1.1 BHM API 11
4.1.1.1 Threads and events 11
4.1.1.2 Parameters 11
4.1.1.3 Diagnostics 11
4.1.1.4 Networking 11
4.1.2 PPM API 11
4.2 Data Endianness 11
4.3 Host Feature Access 11
5 Behavioral Modeling (BHM) API Overview 13
5.1 Interface Definition 13
5.1.1 Parameter definitions 14
5.1.2 Overriding a parameter 15
5.1.3 Bus port definitions 15
5.1.4 Net port definitions 16
5.1.5 Packetnet port definitions 17
5.1.6 Conn (FIFO) port definitions 19
5.2 Initialization 20
5.3 Diagnostic output 21
5.4 Threads 22
5.4.1 Events 23
5.4.1.1 Named Events 24
5.4.2 System Events 24
5.4.3 Delays 24
5.4.3.1 Considerations 25
5.4.3.2 Relationship between delays and time-slice 25
5.5 Callbacks 26
5.5.1 Delays in callbacks 26
5.5.1.1 Notes and Restrictions 29
5.6 Time 30
5.6.1 Current Simulation Time 30

5.6.2 Local Time 30
6 Peripheral Platform Modeling (PPM) API Overview 33
6.1 Bus Slave connection 33
6.1.1 Fixed Mapping 33
6.1.1.1 Aborted Access 36
6.1.2 Dynamic Mapping 36
6.2 Bus master connections 37
6.2.1 By handle 37
6.2.2 By Address Space 38
6.3 Dynamic Bridges 39
6.4 Nets 41
6.5 Conn (FIFO) Support 41
6.5.1 FIFO Word Size 42
6.5.2 Example 42
6.6 Memory Mapped Registers and Bit Fields 42
6.6.1 Callbacks 43
6.6.2 Masking 43
6.6.3 Diagnostics and debug 43
6.6.4 Endian-ness 43
6.6.5 Example 43
6.6.6 Bit-fields 45
6.6.7 Register arrays 46
6.7 Callbacks without registers 48
6.7.1 Overlapping callbacks 49
6.7.2 Fragmented access 50
6.7.3 Simulating a bus error in a callback 50
6.8 Programmers View 51
6.8.1 Automatic Object and Event Generation 51
6.8.2 Objects 51
6.8.2.1 Creating an Object 51
6.8.2.2 Associating Values with Objects 52
6.8.2.3 Removing an Object 52
6.8.3 View Events 52
6.8.3.1 Adding an Event 52
6.8.3.2 Triggering an Event 53
6.8.4 Actions 53
7 Host Code 54
7.1 The constructor 55
7.1.1 Environment Checks 55
7.1.1.1 Raw Intercepts 55
7.1.1.2 ABI Intercepts 55
7.1.2 The Peripheral Simulation Engine ABI 55
7.1.2.1 Raw Intercepts 55
7.1.2.2 ABI Intercepts 55
7.2 Obtaining Intercepted Function Arguments 55
7.2.1.1 Raw Intercepts 55

7.2.1.2 ABI Intercepts ... 56
7.3 Passing the Return Code from an Intercepted Function ... 56
7.3.1.1 Raw Intercepts ... 56
7.3.1.2 ABI Intercepts ... 56
7.4 Data Exchange ... 57
8 Peripheral Model Example ... 58
8.1 Example Source ... 58
8.2 IGEN ... 58
8.2.1 Register model ... 58
8.2.2 Parallel Operations and Signaling Events ... 59
8.2.3 Master Memory Access ... 60
8.2.4 Interrupts ... 60
8.2.5 Behavior using Native host code ... 60
8.3 Platform Overview ... 63
8.3.1 Virtual Platform Design ... 63
8.3.1.1 Virtual Platform Memory Map ... 63
8.3.1.2 Virtual Platform Module Definition ... 63
8.4 Peripheral Model Template ... 64
8.5 Peripheral Model Entry ... 66
8.5.1 Attribute Table ... 66
8.5.2 The Main Function ... 66
8.5.3 Information about available ports ... 67
8.5.4 Information about parameters ... 68
8.5.5 Creating a Slave Port Interface ... 69
8.5.6 Registers ... 69
8.5.6.1 Installing a Register ... 69
8.5.7 Running the Example ... 72
9 Dynamic Slave Port example ... 73
9.1 Instance Peripheral in Module ... 74
9.2 Peripheral iGen Definition ... 74
9.3 Peripheral User Code ... 74
9.3.1 Initialization ... 74
9.3.2 Dynamic mapping ... 75
9.4 Running the Example ... 75
10 QuantumLeap with Peripherals ... 76
10.1 The QuantumLeap Algorithm ... 76
10.2 Example ... 77
10.2.1 Peripheral Code ... 77
10.2.2 Peripheral Native Code ... 78
10.2.3 Module Definition ... 79
10.2.4 Threaded Operation ... 80
10.2.5 Non-Threaded Operation ... 82
10.3 QuantumLeap Results ... 83
10.4 Configuration of Peripheral Native Code ... 83
11 Building Peripherals ... 86
11.1 OVP Library Structure ... 86

11.2 Peripheral Model Makefiles 86
11.2.1 Selecting PSE Type 87
11.3 Building Peripheral Models 87
11.3.1 Building to the Default Output Location 87
11.3.2 Building to a Defined Output Location 88
12 Troubleshooting 89
12.1 Runaway Recursion 89
12.1.1 Error Description 89
12.1.2 Example of Error in Peripheral Intercept Coding 89

1 Preface

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

1.1 Notation

`Code`	A code extract.
`Function`	The name of a function, variable or type
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 peripheral models and peripheral interception libraries (which enable peripheral models to interact with the native host)

Many examples in this document appear in full in the `Examples` directory of your OVP / Imperas installation.

Documents are in the `doc/ovp` and `doc/Imperas` directories.

Refer to `OVP_BHM_PPM_Function_Reference` for details of each function in the API.

2.1 Peripheral Model Template Generation with iGen

Many examples in this document include code that can be generated by the *iGen* productivity tool. The use of *iGen* to generate peripheral model templates is discussed in detail in the document *iGen Peripheral Generator User Guide*.

2.2 Prerequisites

Familiarity with the C language.
If using *iGen*, familiarity with the TCL language.

2.2.1 MinGW

For compilation on a Windows host system the MSYS and MinGW environments are required.
The MinGW program ***make*** is suitable for use with the PSE toolchain and will compile the examples in this document.

2.2.2 Cross Compiler Toolchain

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

3 OVP Simulation Overview

Before creating models for use in the OVP simulation environment you must understand how the components used in that environment interact. This section describes this.

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 a harness or platform to execute binary applications compiled for those processors. It can simulate peripheral models (the subject of this guide) and can be used in 3rd party simulation environments (for example, SystemC). It can be used as a test harness to help validate processor models under construction OVPsim has less functionality than the Imperas CpuManager product.
- *Imperas CpuManager* extends *OVPsim* with a powerful debugger and tools to monitor or analyze a platform as it executes code. It can also be used to create a standalone virtual platform for delivery to your customers.
- *Imperas iGen* is a productivity tool that generates a C template for the model from a TCL programmers view description.

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 model is described in this document with reference to the iGen Peripheral Generator User Guide, the OVP BeHavioral Modeling (BHM) and the Peripheral Programming Model (PPM) APIs.

A peripheral model is compiled into an ELF format executable for a PSE processor architecture. It is dynamically loaded by the simulator *OVPsim* or *CpuManager* or other tools. There are three PSE processor architecture options available, selected by setting the IMPERAS_PSE environment variable in the peripheral model Makefile:

IMPERAS_PSE Setting	PSE Implementation
PSE	32-bit x86 architecture (default)
PSE_RV32	32-bit RISC-V architecture
PSE_RV64	64-bit RISC-V architecture

See section 11.2.1 Selecting *PSE* Type in this document for more information.

4 Behavioral Modeling Methodology

Each instance of a peripheral model runs in its own virtual machine. Each virtual machine has a processor and memory that are separate from any other processors, memories and buses in the platform being simulated; it exists only to execute the code of the peripheral model. This processor is called a Peripheral Simulation Engine or PSE.

Threads and callbacks are initiated in the virtual machine by the simulator. Code in the model relinquishes control back to the simulator by returning from a callback or by calling a function in the API.

A peripheral model can create and control threads of execution using the BHM API and can interact with other components in the platform using the PPM API.

In addition to BHM and PPM, the model can use functions from `libc` for access to the host file system:

- low level i/o: `open(), close(), read(), write()`, etc.
- libc I/O built on the above: `fopen(), fread(), fwrite()`, etc.

It can use `libc` functions that do not use the host:

- string handling: `strcmp(), strcat()`, etc.
- searching and sorting: `qsort()` etc.
- mathematical functions: `sqrt()` etc.

The use of networking, graphics and other I/O routines is prohibited.

4.1 Peripheral Modelling API

Code for the PSE is written using an API that is split into two parts:

4.1.1 BHM API

The BHM API provides general behavioral modeling capabilities.

4.1.1.1 Threads and events

BHM uses a thread-based programming model to:

- Create and delete threads.
- Wait for an amount of simulated time.
- Wait for an *event* in another thread or an external trigger.
- Trigger an *event* in another thread.

4.1.1.2 Parameters

A parameter interface allows different instances of a model to be configured by the platform in different ways.

4.1.1.3 Diagnostics

The model can put messages into the simulator log.

4.1.1.4 Networking

The model can use generic serial and ethernet interfaces

4.1.2 PPM API

The PPM API provides an interface to the platform that instances the model. It models;

- memory local to the peripheral (such a frame store)
- bus interfaces - master or slave – with static or dynamic mappings.
- nets or interrupts
- *packetnets* (for protocols such as CAN, Ethernet or USB)
- named registers with named bit-fields

PPM can produce objects visible to the debugger.
PPM can install callbacks to take control of the simulator in the event of a memory access, change of value on a net or receipt of a packet through a *packetnet.*

4.2 Data Endianness

PSE processors are little-endian. To model a big endian peripheral device the data must be converted at the interface between the peripheral model and each bus interface.

Macros are provided in `ImpPublic/include/target/peripheral/impTypes.h` to convert between little endian and big endian for 2 byte, 4 byte and 8 byte types.

The peripheral model register interface can be programmed to swap data from big to little endian as required. See `ppmCreateNByteRegister()`.

4.3 Host Feature Access

The PSE processor runs in its own private (simulated) memory space, which is isolated from the host environment. To access features of the host system the model writer can create an *intercept library* to be loaded as part of the peripheral model. The intercept library runs on the host so has access to all host features.

As an example, a platform might contain a video display device. The model of the display will represent the display device registers and frame buffer in a PSE but use the host to display the contents of the frame buffer using a video package such as SDL.

Implementation of an interception library is described in Host Code in section 7.

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 which is defined in `ImpPublic/include/target/peripheral/bhm.h`

The BHM provides the ability to write behavioral models of hardware which will interact with other models. To do this a thread-based programming model allows the user to:

- Create and delete threads
- Let a thread wait for an amount of simulated time
- Let a thread wait for an event in another thread
- Perform host I/O

The BHM API gives access to:

- Parameters
- Threads
- Events
- Simulated delays
- Simulator control
- The simulator message stream
- Diagnostic control
- Save/Restore
- A generic serial interface
- A generic Ethernet interface

Code discussed in sections 5.1 "Interface Definition, 5.1.6 "Conn (FIFO) port definitions" and 5.3 "Diagnostic output" can be generated as part of the peripheral template by *iGen*. For more information on using iGen for peripheral model generation please see `iGen_Peripheral_Generator_User_Guide`.

5.1 Interface Definition

This shows how a model defines its name, type, and *parameters*.

A peripheral model must declare a structure describing its interface, which will be interrogated by the simulator (before any peripheral model code is executed). The structure must be called `modelAttrs` and be of type `ppmModelAttr`, defined in `ppm/ppmAttrs.h`. It's at the end of this code block.

```
#include "peripheral/ppm.h"
#include "peripheral/bhm.h"

static ppmParameter parameters[] = {
    {
        .name         = "aNumber",
        .type         = ppm_PT_UNS32,
        .description  = "A description of aNumber",
        .u.uns32Param = { 100, 10, 1000 }
    },
    {
        .name         = "endian",
        .type         = ppm_PT_ENDIAN,
        .description  = "Set the endian"
    },
    {
        .name         = "aString",
        .type         = ppm_PT_STRING,
        .description  = "A description of aString"
    },
    { 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,

    .paramSpecCB      = nextParameter,
    .busPortsCB       = nextBusPort,        // defined in following code example
    .netPortsCB       = nextNetPort,        // defined in following code example
    .packetnetPortsCB = nextPacketnetPort,  // defined in following code example
};

int main() {
    Uns32 aNumber;
    if(bhmUns32ParamValue("aNumber", &aNumber)) {
        bhmPrintf("aNumber = %u\n",  aNumber);
    }

    char aString[128];
    if(bhmStringParamValue("aString", aString, sizeof(aString))) {
        bhmPrintf("aString = %s\n",  aString);
    }

    bhmEndian endian;
    if(bhmEndianParamValue("endian", &endian)) {
        bhmPrintf("endian = %u\n",  endian);
    }
    bhmWaitEvent( bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
    return 0;
}
```

5.1.1 Parameter definitions

`ppmParameter` is a structure filled by the model and read by the simulator. The function `nextParameter` is a callback which must return a pointer to each `ppmParameter` structure in turn, ending with `null`. Each returned structure describes one parameter.

In the peripheral model constructor (in this case, in `main()`) the parameters values can be read using appropriate parameter value functions then used to modify the model's construction or behavior. `ppm.h` defines all parameter types and their access functions.

When the calls to `bhmStringParamValue`() and `bhmEndianParamValue()` in the example are executed, the simulator will search for the parameter value:

1. in an override command on the simulator command line
2. in the module where the peripheral instance is created
3. in modules further up the hierarchy
4. the default value in the peripheral parameter specification

The call will return non-zero if a parameter has been set (cases 1, 2, 3) or zero if not (case 4).

5.1.2 Overriding a parameter

To set a parameter on the simulator command line

- start the simulator with `–showoverrides` ; this will show the full path to each parameter.

```
bash> harness.exe –modulefile module/model.so \
    -showoverrides

...
--override top/module/diagnosticlevel=0 (Uns32) (default=0) (default) ...
--override top/module/aNumber=0 (Uns32) (default=0) (default) ...
--override top/module/aString=0 (String)(default=(null))      ...
--override top/module/endian=0  (Endian)(default=little)      ...
```

- add the appropriate `-override` to the command line

```
bash> harness.exe –modulefile module/model.so \
--override top/module/aNumber=3
```

5.1.3 Bus port definitions

`ppmBusPort` is a structure filled by the model and read by the simulator. The function `nextBusPort()` is a callback which must return a pointer to each `ppmBusPort` structure in turn, ending with `null`. Each returned structure must describe one bus port. In the example the first port is a *slave*, meaning that it must satisfy read or write requests from models elsewhere in the platform.

The *slave bus interface* defines its size in bytes, but the address at which it appear is specified by the module that instances this model. The function `ppmCreateBusSlavePort()` , called from the model's constructor, returns a pointer to a memory region of the same size in the peripheral's address space. This region can be used as memory or have *memory mapped registers* installed in it.

The *master bus interface* defines the maximum and minimum number of address bits that this model must control when working as a bus master. The actual number will be determined by the module that instances this model. The function `ppmOpenAddressSpace()` returns a handle to be used when this model initiates bus reads and writes.

`ppmBusPort`, `ppmNetPort` and `ppmPacketnetPort` have a member `mustBeConnected` which if true requires that the port is connected in the platform, otherwise an error is raised.

```
// (continued from above)

static ppmBusPort busPorts[] = {
    {
        .name            = "sp1",
        .type            = PPM_SLAVE_PORT,
        .addrHi          = 63,
        .mustBeConnected = 1,
        .description     = "sp1 description",
    },
    {
        .name            = "mp1",
        .type            = PPM_MASTER_PORT,
        .mustBeConnected = 1,
        .description     = "mp1 description",
        .addrBitsMin     = 16,
        .addrBitsMax     = 32
    },
    { 0 }
};

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

ppmAddressSpaceHandle mp1;
void                 *sp1;

static void installSlavePorts(void) {
    sp1  = ppmCreateSlaveBusPort("sp1", 64);
}

static void installMasterPorts(void) {
    mp1  = ppmOpenAddressSpace("mp1");
}
```

5.1.4 Net port definitions

`ppmNetPort` is a structure filled by the model and read by the simulator. The function `nextNetPort()` is a callback which must return a pointer to each `ppmNetPort` structure in turn, ending with `null`. Each returned structure must describe one net port.

The first net port is an output, the second an input which calls the model's function `netChangeNotify()`.

```
ppmNetHandle np1, np2;

PPM_NET_CB(netCB){
    bhmPrintf("Net np%u has value %u\n", (Uns32)userData, value);
}

Bool readNP2(void) {
    return ppmReadNet(np2);
}

void setNP1(void) {
    ppmWriteNet(np1, 1);
}

static ppmNetPort netPorts[] = {
    {
        .name            = "np1",
        .type            = PPM_OUTPUT_PORT,
        .handlePtr       = &np1,
        .mustBeConnected = 1,
        .description     = "np1 description"
    },
    {
        .name            = "np2",
        .type            = PPM_INPUT_PORT,
        .handlePtr       = &np2,
        .netCB           = netCB,
        .userData        = 2,
        .mustBeConnected = 1,
        .description     = "np2 description"
    },
    { 0 }
};

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

void writeNP1(Uns32 value) {
    ppmWriteNet(np1, value);
}
```

5.1.5 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.

Packetnet ports can be bidirectional but a model might choose to use the port uni-directionally.

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

```
// define the packet protocol

typedef struct myDataPacketS {
    char header;
    char payload[100];
    char checksum;
} myDataPacket;

// called when a packet is received

static PPM_PACKETNET_CB(pktTrigger) {
    bhmPrintf("packetnet received\n");
}

ppmPacketnetHandle pktPortHandle;

// one packet must be reserved for use by the simulator
myDataPacket packetSharedData;

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

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

// call this to send a packet
void sendPacket(myDataPacket *pkt) {
    ppmPacketnetWrite(pktPortHandle, pkt, sizeof(myDataPacket));
}
```

The variable `packetSharedData`, required by the simulator, reserves space for one packet. 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:
`Examples/Models/Peripherals/packetnet`

5.1.6 Conn (FIFO) port definitions

A *Conn* is an abstraction of a hardware FIFO used for point-to-point links between processors or peripherals. Definition of conn ports is covered in section **Error! Reference source not found.**.

A *Conn* is an abstraction of a hardware FIFO used for point-to-point links between processors or peripherals.

If the model has Conn input ports it must define a callback function using the prototype macro PPM_CONN_INPUT_FN, and set the `connInputsCB` pointer in the `modelAttrs` structure. The `ppmConnInputPort` is a structure filled by the model and read by the simulator. When passed zero, the function should return a pointer to the first `ppmConnInputPort` structure, then each consecutive structure, ending with null when all have been passed.

If the model has Conn output ports it must define a callback function using the prototype macro PPM_CONN_OUTPUT_FN, and set the `connOutputsCB` pointer in the `modelAttrs` structure. The `ppmConnOutputPort` is a structure filled by the model and read by the simulator. When passed zero, the function should return a pointer to the first `ppmConnOutputPort` structure, then each consecutive structure, ending with null when all have been passed.

The `ppmConnInputPort` and `ppmConnOutputPort` structures contains these fields:

Type	**Name**	**Description**
`const char *`	`name`	name of the port
`const char *`	`description`	Short description of the port
`bool`	`mustBeConnected`	True if this port must be connected
`Uns32`	`width`	Width in bits of one word

Example

```
#include "peripheral/ppm.h"

ppmConnInputHandle  port1Handle;
ppmConnOutputHandle port2Handle;

static ppmConnInputPort connInputPorts[] = {
    {
        .name            = "port1",
        .musrBeConnected = 1,
        .handlePtr       = &port1Handle,
        .width           = 32
    },
    { 0 }
};

static ppmConnOutputPort connOutputPorts[] = {
    {
        .name            = "port2",
        .musrBeConnected = 1,
        .handlePtr       = &port2Handle,
        .width           = 32
    },
    { 0 }
};

static PPM_CONN_INPUT_FN(nextConnInputPort) {

    if(!port) {
        port = connInputPorts;
    } else {
        port++;
    }
    return port->name ? port : 0;
}

static PPM_CONN_OUTPUT_FN(nextConnOutputPort) {

    if(!port) {
        port = connOutputPorts;
    } else {
        port++;
    }
    return port->name ? port : 0;
}

ppmModelAttr modelAttrs = {
    // ...
    .connInputPortsCB  = nextConnInputPort,
    .connOutputPortsCB = nextConnOutputPort,
    // ...
};
```

5.2 Initialization

A peripheral model must declare a function `main()`.
At the start of simulation this will be called (with `argc=null` and `argv==0`). 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.
- optionally, wait for the `BHM_SE_END_OF_SIMULATION` event (see System Events in section 5.4.2). If `main()` does not wait for an event the function will complete but any constructed objects will persist
- optionally perform other tasks after the `BHM_SE_END_OF_SIMULATION.`

```
  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 Diagnostic output

A model should produce diagnostic output using `bhmMessage()` and `bhmPrintf()`.

NOTE

When using the default 32-bit x86 architecture, printing *long long ints* using the formats *%llx*, *%lld* and *%llu* in `bhmPrintf` and `bhmMessage` should be avoided due to a bug in the toolchain used to compile the PSE behavioral code. This restriction does not apply to RISC-V architecture PSEs.

In each model the variable usually called `diagnosticlevel` is set by the simulator then used to determine the amount of diagnostic output to be produced. The simulator sets or changes the diagnostic level by calling the callback that was installed using `bhmInstallDiagCB()`

The simulator command line flag `--modeldiags` calls this function.
The `diagnostics` command in the Imperas multiprocessor debugger calls this function.

Bits 0 and 1 of `diagnosticlevel` should be used to determine the level of diagnostics according to these guidelines:

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

Bits 2 and 3 are typically passed to the intercept library part of a peripheral model, if it exists, to control diagnostic output there.

Bit 4 is used by the simulator. When set it causes the simulator to produce diagnostics:

- when a thread is created
- from an ethernet interface
- from an http interface
- when a peripheral input net or packetnet is written
- when a peripheral register is read or written
- when a peripheral aborts during a read or write operation
- from a serial port (for example from a UART)
- when data is written into the peripheral’s address space by another processor

```
#include "peripheral/ppm.h"
#include "peripheral/bhm.h"

//////////////////////////////// Diagnostic level ////////////////////////////////
```

```
// Test this variable to determine what diagnostics to output.

Uns32 diagnosticLevel;

////////////////////////// Diagnostic level callback /////////////////////////

static void setDiagLevel(Uns32 new) {
    diagnosticLevel = new;
}

//////////////////////////////////// Main ////////////////////////////////////

int main(int argc, char *argv[]) {

    diagnosticLevel = 0;
    bhmInstallDiagCB(setDiagLevel);

    if (BHM_DIAG_MASK_LOW(diagnosticLevel)) {
        bhmMessage("I", "DIAG", "Starting model");
    }

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

5.4 Threads

A peripheral model can use a light-weight cooperative threading model to express concurrent behavior. For example a multi-channel DMA engine has several channels operating independently of each other.

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

A thread is given a name and a user data pointer typically used if several copies of the same thread are launched with different contexts.

```
#include "peripheral/bhm.h"

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

typedef struct myThreadContextS {
    const char *name;
    Uns32 wait;
} myThreadContext, *myThreadContextP;

myThreadContext contextA = { "threadA", 1 };
myThreadContext contextB = { "threadB", 2 };;

void myThread(void *user)
{
    myThreadContextP p = user;
    while(1) {
        bhmWaitDelay(p->wait * 1000);
        bhmPrintf("%s says tick\n", p->name);
        bhmWaitDelay (p->wait * 1000);
        bhmPrintf("%s says tock\n", p->name);
    }
}
```

```
int main (void) {
    thA = bhmCreateThread(myThread, &contextA, contextA.name, 0);
    thB = bhmCreateThread(myThread, &contextB, contextB.name, 0);
    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
    return 0;
}
```

Once started, a thread will run to the exclusion of all other simulator activity until it yields so a thread's main loop must yield. Calls which yield are:

- `bhmWaitEvent()`
- `bhmWaitDelay()`

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

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

A thread requires a stack which can be normally allocated by the simulator be setting the 4th argument of `bhmCreateThread()`to zero. If more than 1Mb is required, it must be allocated by the user and the address of its highest location passed to that parameter (since the stack grows downwards). Stack overruns are not detected.

Should it be needed, a thread can determine its own handle using `bhmThisThread()`.

5.4.1 Events

Threads can be 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 restarts it.

```
#include "peripheral/bhm.h"
bhmEventHandle  goEvent;

void myThread(void *user) {
    while(1) {
        bhmWaitEvent(goEvent);
        bhmPrintf("Event has been triggered\n");
    }
}

int main (void) {
    goEvent = bhmCreateNamedEvent("start", "start a transaction");
    bhmCreateThread(myThread, 0, "myThread", 0);
    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
    return 0;
}

void otherThreadOrCallback() {
    bhmTriggerEvent(goEvent);
}
```

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

```
bhmDeleteEvent(start);
```

`bhmDeleteEvent()` returns true when the event handle is valid.

5.4.1.1 Named Events

`bhmCreateNamedEvent()`creates event that works like a regular event, but is also visible to the MPD.

5.4.2 System Events

Two events are created by the simulator and can be accessed using `bhmGetSystemEvent()`. They are:

- `BHM_SE_START_OF_SIMULATION`
- `BHM_SE_END_OF_SIMULATION`

`BHM_SE_START_OF_SIMULATION` is triggered by the simulator after construction and when all the peripherals in the platform have executed code up to their first wait. No application processors will have started at this time.

`BHM_SE_END_OF_SIMULATION` is triggered by the simulator after all the application processors have finished executing code.

```
#include "peripheral/bhm.h"

bhmEventHandle startEvent;
bhmEventHandle endEvent  ;

main() {
    startEvent = bhmGetSystemEvent(BHM_SE_START_OF_SIMULATION)
    endEvent   = bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION)
}

void myThread(void *user){
    bhmWaitEvent(startEvent);
    bhmWaitEvent(endEvent);
}
```

A model must not trigger a system event.

5.4.3 Delays

A thread can pause its execution by waiting for a simulated delay. Alternatively, a thread can schedule the triggering of an event in the future. Triggering an event in the future will cancel any other scheduled triggers of that event.

```
#include "peripheral/bhm.h"

bhmEventHandle  ev1, ev2;

void thread1(void *user) {
    while(1) {
        bhmWaitDelay(120 /*uS*/);
        bhmTriggerEvent(ev1);
        bhmPrintf("thread1 triggers thread2 now\n");
    }
}

void thread2(void *user)
{
    while(1) {
        bhmWaitEvent(ev1);
        bhmTriggerAfter(ev2, 120 /*uS*/);
        bhmPrintf("thread2 triggers thread1 in 120uS\n");
    }
}

int main() {
    ev1 = bhmCreateEvent();
    ev2 = bhmCreateEvent();
    bhmCreateThread(thread1, 0, "thread1", 0);
    bhmCreateThread(thread2, 0, "thread2", 0);
    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));
    return 0;
}
```

The delay is simulated time (not wallclock) specified in microseconds. A fraction of a microsecond may be specified.

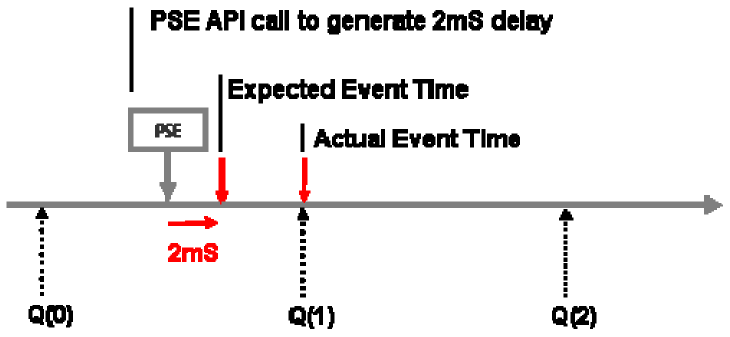
5.4.3.1 Considerations

When setting the simulator timeslice, you must consider the delays that the simulator is expected to model; the timeslice must be set significantly shorter that the shortest delay that is requested by any model.

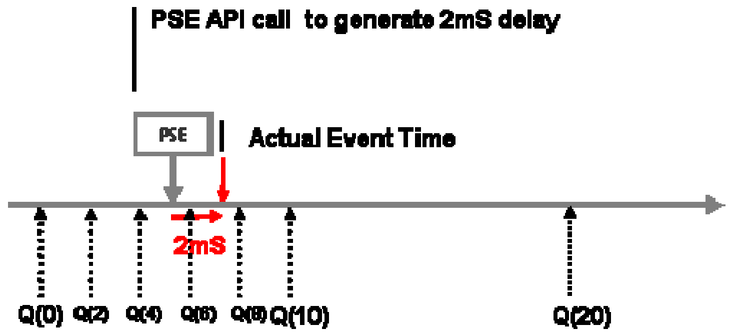
5.4.3.2 Relationship between delays and time-slice

A newly introduced delay will take effect at the start of the next timeslice; consider the following situation:

- The timeslice is set to 10mS.
- 5mS into a timeslice, a register callback in a PSE requests a 2mS delay before triggering an event.
- The current timeslice is in progress so cannot be shortened - other processors have already run instructions in this timeslice on the assumption that it is 10mS.
- The event will trigger as soon a possible - at the start of the next timeslice, i.e. 3mS late.



If however the simulation is running with a timeslice of 1mS then the 2mS requested delay would always fall in the next timeslice and so could be correctly scheduled by the simulator. Note that the *next* timeslice can be shortened because it has not yet started. When a timeslice is shorted, each processor is made to run fewer instructions to compensate.



5.5 Callbacks

Discussed elsewhere in this document, the model can install callbacks to be invoked by the simulator. Callbacks are summarized in this table:

Object with callback	Called when	Delay allowed?
Net	The net is written	no
Packetnet	A packet is transmitted	no
Diagnostic level	Simulator diagnostic level changes	no
View object	The object is written	no
Memory region	The memory is read or written	yes
Memory mapped register	The register is read or written	yes

5.5.1 Delays in callbacks

Peripheral memory region and memory-mapped register callbacks are normally expected to return immediately, without causing any delay or otherwise blocking execution. However, in some circumstances it can be a requirement that such a read or write should cause the initiating processor to block: for example, a peripheral model of a structure like a FIFO might wish to block execution of an initiating processor on a register read if the

FIFO is empty. However, such requests to block execution must not cause the simulation as whole to block (there may be other processors that are still executing) and must not leave the initiating processor in an inconsistent state (where it has partially-executed an instruction), or in a state where it is unable to respond to interrupts.

To support such cases, calls to functions `bhmWaitDelay()`or `bhmWaitEvent()`are supported in callbacks (see table in section 5.5). When they are encountered in a callback, the behavior is as follows:

1. A peripheral thread is implicitly created. The state of the new thread is cloned from the callback thread.

2. The callback thread is cancelled and returns immediately to the initiating processor.

3. The initiating processor is blocked in a state in which, on resumption, it will re-execute the instruction that caused the read or write of the memory-mapped register *from the beginning*. This means that the initiating processor is blocked in a *consistent state*, in which it can respond to interrupts, for example.

4. When the implicitly-created peripheral thread awakens, it runs to its conclusion. Note that it may make further calls to `bhmWaitDelay()`or `bhmWaitEvent()`before terminating.

5. When the implicitly-created peripheral thread terminates, the simulator determines whether the initiating processor is still blocked waiting for the thread. If it is not (it has responded to an interrupt, for example) there is no further action. Otherwise, if it is still blocked, it is restarted, re-executing the peripheral callback.

To use this mechanism to model blocking memory reads and writes, peripheral callbacks must be designed to operate in two modes:

1. A *blocking mode*, in which the complex sequence of actions described above is performed, but the returned data value is ignored;

2. A *non-blocking mode*, which returns data to the initiating processor without delay. Which mode is used depends on the *data ready* state of the peripheral. The required structure of the callback is shown by the following simple example.

A complete example is in: `Examples/Models/Peripherals/blockingCallback`

```
//
// This indicates CONTROL data is available
//
static Bool  CONTROLTready;

PPM_NBYTE_READ_CB(rCONTROL) {

    if(artifactAccess) {

        // no blocking and no side effects
        bport1_REG_data.CONTROL.value = *(Uns32*)data;

    } else {
        if(CONTROLTready) {

            CONTROLTready = False;

            bport1_REG_data.CONTROL.value++;

            // callback requires explicit data transfer
            *(Uns32*)data = bport1_REG_data.CONTROL.value;

        } else {

            // wait for arbitrary delay

            bhmWaitDelay(0);

            CONTROLTready = True;
        }
    }
}

...
//
// Register installer
//
ppmCreateNByteRegister(
    "CONTROL",                      // name
    "Read control",                 // description
    regPort,                        // base
    0,                              // offset from base
    4,                              // bytes
    rCONTROL,                       // readCB
    wCONTROL,                       // writeCB (not shown)
    0,                              // viewCB
    &bport1_REG_data.CONTROL.value,  // address of data (ignored because call
                                    // backs are installed)
    0,                       // userData (unused)
    False,                   // isVolatile
    True,                    // readable, ignored because read CB installed
    True,                    // writable, ignored because write CB installed
    BHM_ENDIAN_BIG           // endian
);
```

This peripheral implements a blocking register called `CONTROL`. The current value is stored in `bport1_REG_data.CONTROL.value` (created by iGen). The mode of the callback is controlled by the `CONTROLTready` Boolean: if `CONTROLTready` is `True`, then the callback operates in non-blocking mode (the first branch of the `if` statement). In this mode, it simply clears `CONTROLTready` and uses the data in variable `bport1_REG_data.CONTROL.value`. If `CONTROLTready` is `False`, it waits for a delta cycle[2] before entering data-ready state and returning.

The boolean `artifactAccess` is used to suppress the blocking behaviour and any side effects that reading the register would otherwise have. This allows debuggers and other diagnostic tools to view the model without perturbing it.

[2] It is possible to wait for a non-zero delay, or for an event instead.

To clarify the operation of the callback, imagine the sequence of operations when a processor reads register `CONTROL` and data is not ready. The operation will be as follows:

1. Execution will go down the *blocking mode* branch (because `CONTROLTready` is `False`), calling `bhmWaitDelay()`.

2. A new peripheral thread will be created, cloning the callback thread state. The callback thread will be cancelled. No transfer of data occurs.

3. The initiating processor will be blocked in a state where it will re-execute the peripheral read on restart.

4. After a delta delay, the peripheral thread will reawaken. It will prepare a new value in `bport1_REG_data.CONTROL.value` and set `CONTROLTready` to `True`. It will then return. Note that the value in `bport1_REG_data.CONTROL.value` is ignored at this point.

5. The initiating processor will be reawakened, and will re-execute the memory-mapped register read, calling function `rCONTROL` again.

6. This time, executing will go down the *non-blocking mode* branch (because `CONTROLTready` is `True`). This will cause a transition back to blocking state (by setting `CONTROLTready` to `False`) and the current value of `bport1_REG_data.CONTROL.value` is copied back to the simulator.

5.5.1.1 Notes and Restrictions

You should be aware of the following when using blocking peripheral callbacks:

1. Only accesses made by processors may be blocked. This is implemented by testing the `artifactAccess` parameter to the callback and doing nothing if it is set. Accesses made by other peripheral callbacks will not block. Attempting to do so will cause the error message:

```
        PSE <name>: <callback_name> may not be used in a nested callback
```

Artifact accesses (from a debugger, for example) will also not block, and the following warning will be printed:

```
        PSE <name>: <callback_name> delay for non-processor access ignored
```

In both cases, the call to `bhmWaitDelay()` or `bhmWaitEvent()` will return the error status `PSE_RR_BADEVENT`.

2. As stated above, returning from an implicitly-created peripheral thread will have no effect on the initiating processor unless it is still blocked waiting for that thread. If it has taken an interrupt and blocked again for a different reason, it will not be restarted, for example.

3. It is possible in a multiprocessor simulation to have several processors concurrently blocked on the same peripheral register. In this case, each initiating processor will have its own implicitly-created thread. In such cases, the peripheral will need to manage the concurrency between the implicitly-created threads so that the initiating processors restart in appropriate fashion. As with real hardware, the peripheral will need information from other sources (e.g. particular values on the address bus) to determine which processor is making each access.

5.6 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 from advancing. While a peripheral thread is running, time will not progress unless the model makes a call to `bhmWaitDelay()` or `bhmWaitEvent()`.

5.6.1 Current Simulation Time

The current simulation time is obtained using the function `bhmGetCurrentTime()`. The current time is advanced at the start of a timeslice so each call to this function within the timeslice will return the same time.

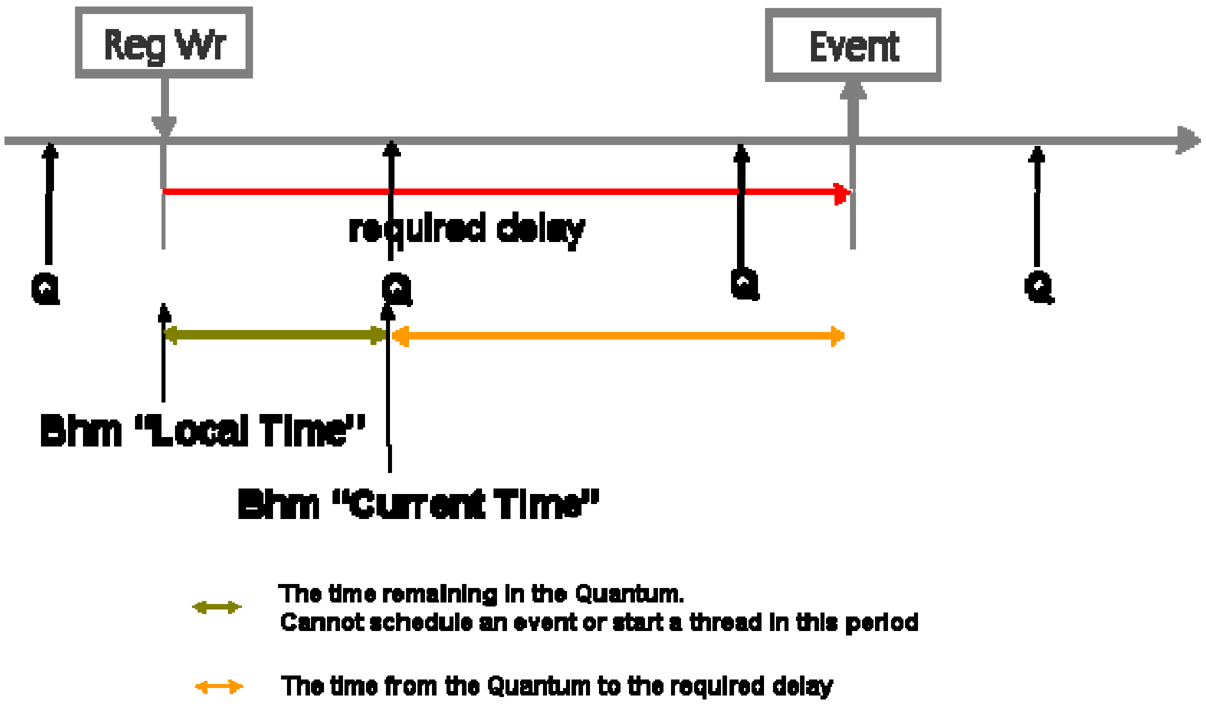
```
#include "peripheral/bhm.h"

int main() {
    Uns64 microseconds1 = bhmGetCurrentTime();

    bhmPrintf("Calling bhmPrintf() does not take time\n");

    Uns64 microseconds2 = bhmGetCurrentTime();

    if( microseconds1 != microseconds2) {
        bhmPrintf("This cannot happen\n");
    }

    bhmWaitDelay(1000);
    Uns64 microseconds3 = bhmGetCurrentTime();

    if(microseconds3 != microseconds1) {
        bhmPrintf("Time has moved on\n");
    }
    return 0;
}
```

5.6.2 Local Time

The function `bhmGetLocalTime()` returns a time in the same format as `bhmGetCurrentTime()` but the time is calculated from the configured MIPS and cycle count of the *initiating processor*, if the peripheral has been activated by a callback from that processor. This will return a time within the current timeslice. If the function is not called from a processor callback then it behaves identically to `bhmGetCurrentTime()`.
Notes

1. Local time is always less than or equal to current time (because current time is advanced before processors are run)
2. Times returned by calls to `bhmGetLocalTime()` do not necessarily increase monotonically in a multiprocessor simulation: if processor A reads a peripheral register towards the *end* of its quantum then a processor B reads a peripheral register towards the *beginning* of the same quantum then local time will appear to go backwards.
3. In a peripheral model, the creation of a thread or an event can only occur on a timeslice boundary. So to request a delay in a peripheral model from inside a callback, it should take into account how much time is left in the quantum.

To do this we use a combination of the simulated time and the local time as shown in the following diagram.


Reg Wr
Event
required delay
Q
Q
Q
Q
Bhm "Local Time"
Bhm "Current Time"
The time remaining in the Quantum.
Cannot schedule an event or start a thread in this period
The time from the Quantum to the required delay

The callback determines the delay required when the processor performs the write. The use of `bhmTriggerAfter(delayStart, usec_wait)` is used to cause a peripheral model thread to perform the required action after the delay has expired:

This example calculates the time to the end of the current timeslice and adds this to the required delay of 3000uS. The thread reports the exact time it wakes up and shows how the timeslice length was adjusted to compensate.

```
bhmEventHandle delayStart;

void periodicThread(void* user) {

    if(PSE_DIAG_LOW) {
        bhmMessage("I", "THR", "Initialized");
    }

    while(1) {
        bhmWaitEvent(delayStart);

        double flt_sim_now = bhmGetCurrentTime();
        double flt_cpu_now = bhmGetLocalTime();
        bhmMessage("I", "THR", "Thread wakes up\n");
        bhmMessage("I", "THR", "CurrentTime : %10.3f\n", flt_sim_now);
        bhmMessage("I", "THR", "LocalTime   : %10.3f\n", flt_cpu_now);
    }
}

PPM_NBYTE_WRITE_CB(writeR1) {

    double flt_sim_now = bhmGetCurrentTime();
    double flt_cpu_now = bhmGetLocalTime();
    double request     = 3000;

    bhmMessage("I", "DEL", "CurrentTime     : %10.3f\n", flt_sim_now);
    bhmMessage("I", "DEL", "Proc   Time     : %10.3f\n", flt_cpu_now);
    bhmMessage("I", "DEL", "Required delay  : %10.3f\n", request);

    double usec_wait = request - (flt_sim_now - flt_cpu_now);

    bhmMessage("I", "DEL", "Requested delay : %10.3f\n", usec_wait);

    if (usec_wait >= 0) {
        bhmMessage("I", "DEL",  "Wait Delay : %10.3f usec \n", usec_wait);
        bhmTriggerAfter(delayStart, usec_wait);

    } else {
        if(PSE_DIAG_MEDIUM) {
            bhmMessage("W", "ERR",
                 "Delay %10.3f usec falls within slice",
                 usec_wait
            );
        }
    }
}

PPM_CONSTRUCTOR_CB(constructor) {
    delayStart = bhmCreateEvent();

    bhmCreateThread(periodicThread, 0, "periodicThread", 0);
    periphConstructor();
}
```

6 Peripheral Platform Modeling (PPM) API Overview

The PPM API allows the model to interact with components in a platform. PPM includes operations to:

- Bind to a bus port which is connected to a bus.
- Bind to a net port which is connected to a net.
- Read and write to memory through bus ports.
- Generate and receive interrupts through net ports.
- Create and control windows into other address spaces
- Install callbacks on memory regions
- Create *memory mapped registers*.
- Create *programmers view* objects that can be seen by the debugger

The PPM API is defined in the header file:
`ImpPublic/include/target/peripheral/ppm.h` and implemented in the library
`lib/$IMPERAS_ARCH/TargetLibraries/pse-elf/libImperas.a.`
These file are automatically included by the peripheral compiler and linker.

The definition of bus, net and packetnet ports is covered in section 5.1.

6.1 Bus Slave connection

The model can connect to a simulated bus as a slave in three ways:

1. By mapping an area of the peripheral model address space to the address space of the simulated bus. The two regions must be of the same size. Refer to `ppmCreateSlaveBusPort`(). This is used to model a peripheral that contains memory this is visible in the programmer’s view but to which reading and writing has no side effects.

2. By requesting read/write callbacks on the address space. Refer to `ppmInstallReadCallback`(), `ppmInstallWriteCallback`() and `ppmInstallChangeCallback`(). This is used when reading and writing the memory has side effects.

3. By creating *memory mapped register* objects in the address space. A register can have read/write callbacks to allow side affects to occur in the model when the register is accessed. See `ppmCreateNByteRegister`().

6.1.1 Fixed Mapping

In a fixed mapping the address of the slave port is specified in the platform, not in the model. The address offset specified as `loaddress` in the bus slave port connection is not visible through this API; an access to the lowest address on the simulated bus appears at lowest address in the mapped region.

```
#include "peripheral/bhm.h"
```

```
#include "peripheral/ppm.h"

#define  sizeInBytes 32

static ppmBusPort busPorts[] = {
    {
        .name            = "sp1",
        .type            = PPM_SLAVE_PORT,
        .addrHi          = sizeInBytes-1,
        .mustBeConnected = 1,
        .description     = "sp1 description",
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

Uns8 mappedRegion[sizeInBytes];   // a region to be read/written

int main() {

    ppmOpenSlaveBusPort(
        "sp1",
        mappedRegion,
        sizeof(mappedRegion)
    );

    // code in the peripheral can read or write mappedRegion[]
    // which is now shared with an other models in the platform

    bhmPrintf("memory=%u\n", mappedRegion[0]);
    return 0;
}
```

A read or write by a processor to the region specified in the bus slave port connection will 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, it can install callbacks on `mappedRegion`. A read or write to the port will trigger a callback.

Callback arguments include:

- `addr`: the address of the access in the peripheral's address space. This can be converted to an offset by subtracting the address of `mappedRegion`.
- `bytes:` the size of the access

- `artifactAccess:` true if the access comes from the debugger
- `data`: (in the write callback)

The value returned by the read will be that returned by `readCallback()`.

```
#include "peripheral/bhm.h"
#include "peripheral/ppm.h"

#define  sizeInBytes 32

static ppmBusPort busPorts[] = {
    {
        .name            = "sp1",
        .type            = PPM_SLAVE_PORT,
        .addrHi          = sizeInBytes-1,
        .description     = "sp1 description",
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

Uns8 mappedRegion[sizeInBytes];   // a region to be read/written

PPM_READ_CB(readCallback) {
    bhmPrintf(
        "readCallback:  offset:%u  size:%u  %s\n",
        (Uns32)(((Uns8*)addr)-mappedRegion),
        bytes,
        artifactAccess ? "artifact" : "regular"
    );
    return 99;
}

PPM_WRITE_CB(writeCallback) {
    bhmPrintf(
        "writeCallback:  offset:%u  size:%u  data:%u   %s\n",
        (Uns32)(((Uns8*)addr)-mappedRegion),
        bytes,
        data,
        artifactAccess ? "artifact" : "regular"
    );
 }

int main() {

    ppmOpenSlaveBusPort(
        "sp1",
        mappedRegion,
        sizeof(mappedRegion)
    );

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

    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));

    return 0;
}
```

6.1.1.1 Aborted Access

The peripheral can model incomplete reads or writes by calling `ppmReadAbort()` or `ppmWriteAbort()` from inside the callback. If it's in the correct mode, the processor that initiated the access will take an exception.

6.1.2 Dynamic Mapping

Rather than taking the bus port address from the platform, a model can specify its own address on the simulated bus and perhaps change this mapping during simulation.
The bus port specification must set the remappable field to true.
Any port address supplied by the platform will be ignored.

```
#include "peripheral/bhm.h"
#include "peripheral/ppm.h"

#define  sizeInBytes 32
#define  portName    "sp1"

static ppmBusPort busPorts[] = {
    {
        .name            = portName,
        .type            = PPM_SLAVE_PORT,
        .addrHi          = sizeInBytes-1,
        .description     = "sp1 description",
        .remappable      = 1
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

Uns8 mappedRegion[sizeInBytes];       // a region to be read/written

Uns32 addr1 = 0x40000000;
Uns32 addr2 = 0x50000000;
```

```
int main() {
    ppmCreateDynamicSlavePort(        // set the initial port address
        portName,
        addr1,
        sizeInBytes,
        mappedRegion
    );

    bhmWaitDelay(100);

    ppmDeleteDynamicSlavePort(        // remove the old mapping
        portName,
        addr1,
        sizeInBytes
    );

    ppmCreateDynamicSlavePort(        // remap
        portName,
        addr2,
        sizeInBytes,
        mappedRegion
    );

    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));

    return 0;
}
```

6.2 Bus master connections

A bus master port lets the peripheral model initiate bus transactions. There are two methods of connecting a model to its bus master port:

6.2.1 By handle

Get a handle to the address space on the simulated bus and then use `ppmReadAddressSpace()` and `ppmWriteAddressSpace()` to read or write to the bus.

This method gives the peripheral access to any size of address space but has the overhead of an intercepted function on each access.

```
#include "peripheral/bhm.h"
#include "peripheral/ppm.h"

#define  portName    "mp1"

static ppmBusPort busPorts[] = {
    {
        .name            = portName,
        .type            = PPM_MASTER_PORT,
        .addrBits        = 32,
        .description     = "mp1 description",
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

ppmAddressSpaceHandle mh;

int main() {

    mh = ppmOpenAddressSpace(portName);

    Uns32 data;
    Addr  address = 0x10000000;

    bhmWaitDelay(100);

    ppmReadAddressSpace (mh, address, sizeof(data), &data);
    data++;
    ppmWriteAddressSpace(mh, address, sizeof(data), &data);

    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));

    return 0;
}
```

6.2.2 By Address Space

Map an area of PSE 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 each bus access does not require a function to be intercepted, The simulator cannot track bus activity caused by the model. The peripheral model has access to range of addresses limited to the size of the window.

```
#include "peripheral/bhm.h"
#include "peripheral/ppm.h"
#include <string.h>

#define  portName    "mp1"

static ppmBusPort busPorts[] = {
    {
        .name            = portName,
        .type            = PPM_MASTER_PORT,
        .addrBits        = 32,
        .description     = "mp1 description",
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

ppmExternalBusHandle mh;
static char window[128];

int main() {

    Addr remoteAddress = 0x10000000;

    mh = ppmOpenMasterBusPort(
        portName,
        window,
        sizeof(window),
        remoteAddress
    );

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

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

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

    bhmWaitEvent(bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION));

    return 0;
}
```

6.3 Dynamic Bridges

A common platform requirement is for an address map 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 can map (or remap) address regions from one bus to another using a *dynamic bridg*e. A dynamic bridges is 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 with a 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. When remapping, the peripheral model must keep track of active bridges and delete an old bridge before a new one is 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 "peripheral/bhm.h"
#include "peripheral/ppm.h"

#define  portName1    "p1"
#define  portName2    "p2"

static ppmBusPort busPorts[] = {
    {
        .name            = portName1,
        .type            = PPM_MASTER_PORT,
        .addrBits        = 32,
        .description     = "p1 description",
    },
    {
        .name            = portName2,
        .type            = PPM_MASTER_PORT,
        .addrBits        = 32,
        .description     = "p2 description",
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {
    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

int main() {

    Addr  slavePortLoAddress  = 0x10000000;
    Uns32 windowSizeInBytes   = 0x1000;
    Addr  masterPortLoAddress = 0;

    // create bridge
    ppmCreateDynamicBridge(
        portName1,
        slavePortLoAddress,
         windowSizeInBytes,
        portName2,
        masterPortLoAddress
    );

    bhmWaitDelay(100);

    // delete bridge
    ppmDeleteDynamicBridge(
        portName1,
        slavePortLoAddress,
         windowSizeInBytes
    );

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

6.4 Nets

A net is usually used to model a wire carrying a logic value. A net carries an integer value which can represent anything you wish but usually the values zero or non-zero represent logic values zero or one. A peripheral model can declare a net port in its interface; the platform must declare the net and connect it to ports. The example in section 5.1.4 shows the use of net input and output ports.

`handlePtr` must be set to point to a net handle. During platform construction the simulator sets the value of the handle which can then be used to read or write the net. If the net is an input, netCB can be set to a callback function which will be called when the net is written.

The value of a net is read using `ppmReadNet()`. It is written using `ppmWriteNet().`

Writing to a net will call the callback in all models that has one registered, even if the current value is re-written. It is the responsibility of the model to suppress writing the same value multiple times. The order in which other models are called cannot be guaranteed.

A net connection can be passed through the module hierarchy; this will not affect simulation performance.

The net callback must not block or delay; it must store the new value and/or trigger an event then return.

`ppmOpenNetPort()` is a deprecated method of obtaining a net handle and `ppmInstallNetCallback()` a method of installing a callback.

6.5 Conn (FIFO) Support

A *Conn* is an abstraction of a hardware FIFO used for point-to-point links between processors or peripherals. The definition of conn or FIFO ports is covered in section **Error! Reference source not found.**.

A peripheral model can put data into a FIFO or read data out using a polling or event driven interface and can query a FIFO to determine its dimensions, connections and how much data is currently in the queue. Please refer to the section on FIFOs in the OVP_BHM_PPM_Function_Reference document or the example below.

6.5.1 FIFO Word Size

The FIFO interface is intended to model a hardware FIFO, so sends one word at a time. A word is specified in bits in the `width` member of the `ppmConnInputPort` and `ppmConnOutputPort` structures (see section 5.1.6) rounded up to the nearest byte (1-8 bits = 1 byte, 9-16 bits = 2 bytes). This number controls how many bytes are transferred by each call to `ppmConnGet()` and `ppmConnPut()`. In the current implementation the sender and receiver must use the same word size.

6.5.2 Example

```
Directory: Examples/Models/Peripherals/FIFO
```

This example uses a FIFO to communicate (in one direction) between two identical peripheral models, one configured as a source and one a sink of character data. Referring to the above directory the `module` directory contains a module described in TCL that creates two instances of the peripheral model connected by a FIFO. The module is built using iGen and the host C compiler.

The `peripheral` directory contains the peripheral model. Its structure is described in TCL, the behaviour in the C file `user.c`. It is built using iGen and the PSE C compiler.

In `user.c` the function `constructor` checks the FIFO input and output ports to determine if this instance is to be used as the source or sink of data. Data will be read or written to the FIFO using `ppmConnGet()` and `ppmConnPut()` which are non-blocking (polling) functions.

To prevent the peripheral model from wasting CPU time by polling, a FIFO input port can be bound to a peripheral event using `ppmRegisterConnInputEvent()` or a FIFO output port can be bound to an event using `ppmRegisterConnOutputEvent()`. Functions `readFromFifo()` and `writeToFifo()` in `user.c` in the example do this. Note that `ppmConnGet()` can be used to peek at the first word in the FIFO without removing it.

6.6 Memory Mapped Registers and Bit Fields

To model memory mapped registers, a region of memory in the peripheral model’s address space (the *window*) must be reserved then mapped to the simulated bus using `ppmCreateSlaveBusPort()`. Registers are then installed with different offsets from the base of the window using `ppmCreateNByteRegister()`. Each register has a name and description, a separate region when it’s data is stored, plus optional read, write and view callbacks. It is accessed when an application processor (or other bus master) reads or writes to an address on the simulated bus that is mapped to the window in the peripheral model.

Without `readCB`, `writeCB` or `viewCB` callbacks, data will be read or written to the storage referenced by the `data` pointer. A read or write that is larger than the `bytes` parameter is illegal. A read or write of fewer bytes will access a lower part of the storage.

6.6.1 Callbacks

If a read or write requires side effects then the `readCB` and or `writeCB` must be supplied in which case data is not automatically copied to the storage.
Note that more than one register can share a callback; the `userData` pointer can be used to distinguish which register was accessed. If at any time the true value of the register is not stored in the location referenced by `userData`, then `viewCB` must also be supplied.

6.6.2 Masking

Masking of bit fields during reads and writes can be implemented by the simulator when required. See `ppmCreateRegisterField.`

6.6.3 Diagnostics and debug

Reads and writes to the register will trigger debugger event-points and (if the model's diagnostic level is set to enable system diagnostics) cause a message to be sent to the simulator log.

6.6.4 Endian-ness

The simulator can be programmed to byte-swap data supplied to and from the callbacks (if supplied) or byte-swap the data as it is read or written to the register's storage.

6.6.5 Example

In the example, `reg1` occupies the first 4 bytes of the 32-byte port. The contents of the variable `reg1` will always be the register's value.
`reg2` occupies the next 4 bytes. It's read, write and view functions are handled by callbacks which cause side effects.
The remaining 24 bytes of the window have no visible registers but will appear on the simulated bus, so can be accessed by code in the peripheral.

```
#include "peripheral/ppm.h"
#include "peripheral/bhm.h"

static ppmBusPort busPorts[] = {
    {
        .name            = "regPort",
        .type            = PPM_SLAVE_PORT,
        .addrBits        = 32,
        .description     = "register port",
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {
    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

Uns32 reg1, sideEffect=0;

// this is cleared by another thread (not shown)
Bool  dataReady;

PPM_NBYTE_READ_CB(readCB) {

    // validate the access by checking bytes, offset etc,
    if(bytes != sizeof(sideEffect)) {
        bhmMessage("E", "ERR", "Illegal read. Wrong number of bytes\n");
        return;
    }
    // return the data, with side effects
    *(Uns32*)data = sideEffect++;
}

PPM_NBYTE_WRITE_CB(writeCB) {

    // validate the access by checking bytes, offset etc,
    if(bytes != sizeof(sideEffect)) {
        bhmMessage("E", "ERR", "Illegal write. Wrong number of bytes\n");
        return;
    }
    sideEffect = *(Uns32*)data;
    dataReady  = True;
}

// (Could be achieved without a callback in this case)
PPM_NBYTE_VIEW_CB(viewCB) {
    *(Uns32*)data = sideEffect;
}

int main (){

    void *regPort = ppmCreateSlaveBusPort("regPort", 32);

    ppmCreateNByteRegister(
        "reg1",                 // name
        "control register1",    // description
         regPort,               // base of window
         0,                     // offset from window base
         sizeof(reg1),          // size in bytes
         0,                     // bus read function
         0,                     // bus write function
         0,                     // debugger view function
        &reg1,                  // storage
         0,                     // userData
         True,                  // volatile register
         True,                  // read access
         True,                  // write access
         BHM_ENDIAN_LITTLE
    );
    ppmCreateNByteRegister(
        "reg2",                 // name
        "control register2",    // description
         regPort,               // base of window
         4,                     // offset from window base
         sizeof(sideEffect),    // size in bytes
         readCB,                 // read function
         writeCB,                // write function
         viewCB,                      // debugger view function
        &sideEffect,             // storage
         0,
         True,                   // volatile register
         True,                   // read access
         True,                   // write access
         BHM_ENDIAN_LITTLE
    );

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

6.6.6 Bit-fields

Bit fields give names to groups of bits in a register and allow control of how bits are read or written:

```
// (continuing the previous example)

    registerHandle reg1 = ppmCreateNByteRegister(
        "reg1",                  // name
        "control register1",     // description
         regPort,                // base of window
         0,                      // offset from window base
         sizeof(reg1),           // size in bytes
         0,                      // bus read function
         0,                      // bus write function
         0,                      // debugger view function
        &reg1,                   // storage
         0,                      // userData
         True,                   // volatile register
         BHM_ENDIAN_LITTLE
    );
    ppmCreateRegisterField(
        reg1,                    // containing register
        "f1",                    // name
        "f1 description",        // description
         0,                      // offset from LSB
         4,                      // number of bits
         True,                   // can be read
         True                    // can be written
    );
    ppmCreateRegisterField(
        reg1,                    // containing register
        "f2",                    // name
        "f2 description",        // description
         4,                      // offset from LSB
         2,                      // number of bits
         True,                   // can be read
         False                   // cannot be written
    );
```

When a register with no write callback is written, bitfields without write access will not be changed. When a register with no read callback is read, bitfields without read access will be read as zero.

6.6.7 Register arrays

Register functions accept a `userData` pointer which is passed to the callbacks. This allows arrays or banks of registers to share functionality. This example shows a pair of registers replicated 8 times.

```
#include "peripheral/ppm.h"
#include "peripheral/bhm.h"
#include <stdio.h>

static ppmBusPort busPorts[] = {
    {
        .name            = "regPort",
        .type            = PPM_SLAVE_PORT,
        .addrBits        = 64,
        .description     = "register port",
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,
    .busPortsCB    = nextBusPort
};

#define BANKS    8

typedef struct bankTypeS {

    Uns32 control;
    Uns32 data;
    Bool  dataReady;

} bankType, *bankTypeP;

bankType bank[BANKS];

PPM_NBYTE_READ_CB(readCB) {

    bankTypeP p = userData;

    // validate the access by checking bytes, offset etc,
    if(bytes != sizeof(p->data)) {
        bhmMessage("E", "ERR", "Illegal read. Wrong number of bytes\n");
        return;
    }
    // return the data, with side effects
    *(Uns32*)data = p->data++;
}

PPM_NBYTE_WRITE_CB(writeCB) {
    bankTypeP p = userData;

    // validate the access by checking bytes, offset etc,
    if(bytes != sizeof(p->data)) {
        bhmMessage("E", "ERR", "Illegal write. Wrong number of bytes\n");
        return;
    }
    p->data       = *(Uns32*)data;
    p->dataReady  = True;
}

int main (){

    void *regPort = ppmCreateSlaveBusPort("regPort", 64);

    Uns32 i;
    for(i=0; i < BANKS; i++) {

        char name[8];
       sprintf(name, "control%u", i);

       registerHandle reg = ppmCreateNByteRegister(
             name,                    // name
             "control register",      // description
              regPort,                // base of window
              0,                      // offset from window base
              sizeof(bank[i].control), // size in bytes
              0,                      // bus read function
              0,                      // bus write function
              0,                      // debugger view function
             &bank[i].control,        // storage
              0,                      // userData
              True,                   // volatile register
              True,                   // read access
              True,                   // write access
              BHM_ENDIAN_LITTLE
       );

       sprintf(name, "data%u", i);

       ppmCreateNByteRegister(
             name,                    // name
             "data register",         // description
              regPort,                // base of window
              4,                      // offset from window base
              sizeof(bank[i].data),   // size in bytes
              readCB,                 // read function
              writeCB,                // write function
              0,
              &bank[i].data,          // storage
              &bank[i],               // userData
              True,                   // volatile register
              True,                   // read access
              True,                   // write access
              BHM_ENDIAN_LITTLE
       );

       ppmCreateRegisterField(
             reg,                     // containing register
             "f1",                    // name
             "f1 description",        // description
              0,                      // offset from LSB
             4,                        // number of bits
             True,                     // can be read
             True                      // can be written
       );

       ppmCreateRegisterField(
            reg,                       // containing register
            "f2",                      // name
            "f2 description",          // description
             4,                        // offset from LSB
             2,                        // number of bits
             True,                     // can be read
             False                     // cannot be written
       );
    }

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

6.7 Callbacks without registers

We have discussed how to map part of the PSE address space to a slave port and how to install registers with callback in that space. It is also possible to install a callback without creating a register. This is useful when reading or writing to a range of addresses will have side effects but the peripheral does not have conventional registers at these addresses.

```
Uns32 SIZE = 4;

PPM_NBYTE_READ_CB(readCB) {
    const char *txt = userData;
    bhmPrintf("Read %s offset:%u    bytes:%u\n", txt, offset, bytes);
}

PPM_NBYTE_WRITE_CB(writeCB) {
    const char *txt = userData;
    bhmPrintf("Write %s offset:%u    bytes:%u\n", txt, offset, bytes);
}

PPM_CONSTRUCTOR_CB(constructor) {
    void *window = ppmCreateSlaveBusPort("sp1", SIZE);

    ppmInstallNByteCallbacks(
        readCB,                // the callback
        writeCB,               // optional user data
        "data",
        window,                // address of the port window
        SIZE,                  // size of the port window
        1,
        1,
        0,
        BHM_ENDIAN_LITTLE
    );
}
```

In the read callback (prototype defined in the macro `PPM_NBYTE_READ_CB`) the parameter `offset` is the address of the access in the window that triggered the call. `data` points to where the read data should be copied to by this function. `bytes` is the size of the access in bytes, `userData` is user specific data that was passed to the install function. `artifactAccess` is true if the access is a simulation artifact. This occurs when the simulator is pre-fetching values for dynamic code translation or can be caused by a debugger. The model must inhibit side effects for this kind of access.

Parameters to the write callback are the same as for read except that `data` points to where the data is coming from.

`ppmInstallNByteCallbacks()` installs the supplied callbacks. If the read function is null then no callback is installed. If the `readable` parameter is true, a read access to the port will read the region referenced by `window`. If `readable` is false, there will be a bus error.

If the write function is null then no callback is installed. If the `writable` parameter is true, a write access to the port will write to the region referenced by `window`. If `writable` is false, there will be a bus error.

6.7.1 Overlapping callbacks

If `ppmInstallNByteCallbacks()` is used more than once with overlapping regions, the more recently installed callback will be called. This also applies when a mixture of callbacks and registers are installed. Therefore a read and write callback can be installed on a region, then registers installed on part of the region. The previously installed callbacks will then catch any reads or writes that fall between the registers.

```
Uns32 SIZE = 8;

PPM_NBYTE_READ_CB(readCB) {
    bhmPrintf("Warning. Read: offset:%u    bytes:%u\n", offset, bytes);
}

PPM_NBYTE_WRITE_CB(writeCB) {
    bhmPrintf("Warning. Write: offset:%u    bytes:%u\n", offset, bytes);
}

PPM_CONSTRUCTOR_CB(constructor) {
    void *window = ppmCreateSlaveBusPort("sp1", SIZE);

    ppmInstallNByteCallbacks(
        readCB,                 // the callback
        writeCB,                // optional user data
        0,
        window,                 // address of the port window
        SIZE,                   // size of the port window
        1,
        1,
        0,
        BHM_ENDIAN_LITTLE
    );

    Uns8 dr;

    ppmCreateNByteRegister(
        “dr”,                   // name
        "data register",        // description
        window,                 // base of window
        4,                      // offset from window base
        1,                      // size in bytes
        readCB,                 // read function
        0,                      // write function
        0,                      // view function
        &dr,                    // storage
        0,                      // userData
        False,                  // (not) volatile
        True,                   // read access
        False,                  // no write access
        BHM_ENDIAN_LITTLE
    );
}
```

In this example an 8 byte address region is mapped to the slave port `SP`. A callback is installed on the whole region, then a register installed over one byte. A one-byte read from the port with offset 4 will read from the register `dr ;` a read from elsewhere will print a warning. For example a 4-bye read from with offset 4 will read the register then read the remaining 3 bytes from the region mapped to `readCB()` which produces a warning. Section 6.7.2 explains why this happens.

6.7.2 Fragmented access

A read or write to memory with a range that straddles more than one region with be split by the simulator into multiple accesses.

6.7.3 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()`

6.8 Programmers View

The programmers view provides additional information that can be accessed by the Imperas Professional tools.

A peripheral model can be written to provide information to the simulator about its internal operation that can be accessed by the Imperas MPD and by intercept libraries when the model source is not available. Use of the programmers view includes:

- providing a view of arbitrary objects within the model
- defining special actions that the model can perform, for example flushing a buffer or resetting part of the model.
- allowing internal values to be read from the model
- generating *eventpoints* (stopping in the debugger when an event is triggered) on specific condition being met, for example data received or buffer overflow.

6.8.1 Automatic Object and Event Generation

Some PPM functions automatically generate information for the programmers view. For example a register created using the `ppmCreateNByteRegister()`creates a register object that can be accessed by the Imperas MP Debugger and also creates read and write *events* that can trigger a breakpoint in the MPD or be detected by an intercept library. See section 8.5.6.

6.8.2 Objects

A view object can be explicitly added using the `ppmAddViewObject()` function.

View objects are hierarchical; the initial object will be supplied with `NULL` parent indicating that it is the top-level object, subsequent objects can be added within previously added objects to build a tree.

When an object is read by the debugger or an intercept library, the value may be provided as a:

- variable
- constant value
- 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.

6.8.2.1 Creating an Object

This example makes a two-level hierarchy:

```
ppmViewObject topObject = ppmAddViewObject(
    NULL,                           // Top level so parent is null
    "topObject",                    // object name, visible in the debugger
    "top object description"        // optional description
);
```

```
ppmViewObject secondLevelObject = ppmAddViewObject(
    topObject,                      // parent is previous object
    "level2Object",                 // object name, visible in the debugger
    "level 2 object description"    // optional description
);
```

6.8.2.2 Associating Values with Objects

The value of the object `level2Object` is accessed using a callback function so is defined using the function `ppmSetViewObjectValueCallback()`.

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

`readControlValueCB` is a function that must return the current value of the object.

A new object is created and associated with a variable that indicates its state:

```
ppmViewObject thirdLevelObject = ppmAddViewObject(
     viewControlReg,
     "level3Object",
     "level 3 Object description"
);

Bool level3Value;

ppmSetViewObjectRefValue(thirdLevelObject, PPM_VVT_BOOL, &level3Value);
```

6.8.2.3 Removing an Object

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

```
ppmDeleteViewObject(thirdLevelObject);
```

6.8.3 View Events

6.8.3.1 Adding an Event

An event can be added into the peripheral model to allow any occurrence to be signaled to the MPD or an intercept library. An event can be generated at the top-level or can be attached to an existing view object. It's place in the hierarchy does not affect it's behaviour, just where it appears.

This shows two events, one created at the top level and one associated with an object.

```
//
// Create events which can trigger eventpoints
//
interruptEvent = ppmAddViewEvent(
    NULL,                             // Top level
    "reset",                         // Event name
    "triggered when a reset occurs"    // Event description
);

overflowEvent = ppmAddViewEvent(
        thirdLevelObject,
        "overflow",
        "triggered when overflow occurs"
);
```

6.8.3.2 Triggering an Event

An event can be explicitly triggered in a peripheral model using `ppmTriggerViewEvent()`at any point in the behavior of the peripheral model, as shown in the following example taken from the behavior of the counter register in the peripheral model. When the counter overflows, an interrupt is raised and the `wrapEvent` is triggered. Anything waiting on or having a breakpoint set on this event will be triggered.

```
void updateCounter() {
    counter++;
    if ( counter == 0 ) {
        // Counter just overflowed. Generate interrupt.
        generateInterrupt();

        // Inform simulator of overflow.
        ppmTriggerViewEvent(overflowEvent);
    }
}
```

6.8.4 Actions

A view object can be used to cause an arbitrary action can in a peripheral model. An action function is registered using `ppmAddViewAction()`.
The following code allows the user of the MPD to reset a counter – an operation that this model does not normally allow.

```
void resetCounterActionCB(void *userData) {
    counter = 0;              // counter reset to zero without an interrupt
}

ppmAddViewAction(
    thirdLevelObject,         // Parent view object
    "reset",
    "reset the timer counter",
    resetCounterActionCB,
    0                             // user data
);
```

7 Host Code

The PSE is an isolated environment that runs a peripheral model. Access to the host environment is restricted; system functions `open()`, `close()`, `read()`, `write()` and `fstat()` (and their libc buffered equivalents `fopen`() etc.) are available. Should a model require access to other system functions, the user can use *binary interception* to run code on the host. Host functions must be used with care; a blocking host function will block the simulator.

There is a comprehensive description of function interception applied to application code in `OVP_VMI_OS_Support_Function_Reference`. This section illustrates the use of binary interception to link between PSE code and 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 loaded with the peripheral instance. This is called an *intercept library*. The intercept library API binds by name functions in the PSE to functions in the intercept library.

Empty *stub* functions in the PSE are called when host functionality is required. When the peripheral calls the stub function, control passes to a callback in the intercept library. Since the intercept library can be linked with other host libraries the peripheral has access to any functionality that the host computer can support. The interface between stub functions and host functions in the library can be specified on one of two ways:

1. Using *raw* intercept functions. With these, the host code is responsible for extracting arguments from known registers using the PSE call ABI (see section 7.2 below).
2. Using *ABI* intercept functions. With these, the host code is presented a processed argument list in a useful form. The ABI intercept function methodology should be used where possible because it is simpler and more intuitive.

To make the simulator load an intercept library, set the `extension` field in the peripheral `modelAttrs` structure to the name of library (without its file extension) and put the library in the same directory as the peripheral model executable.

7.1 The constructor

7.1.1 Environment Checks

7.1.1.1 Raw Intercepts

When using *raw* intercepts, a peripheral model intercept library should check that it is running on an appropriate PSE processor type (for example, `pse` or pse_`RV32`):

```
const char *procType = vmirtProcessorType(processor);

if (strcmp(procType, "pse") != 0) {
    vmiMessage("F", PREFIX, "Processor must be a PSE\n");
}
```

7.1.1.2 ABI Intercepts

When using *ABI* intercepts, a PSE processor type check is usually not required because the intercept library has no ABI dependency.

7.1.2 The Peripheral Simulation Engine ABI

7.1.2.1 Raw Intercepts

When using *raw* intercepts, the peripheral model must explicitly access function arguments using the known ABI of the PSE. Depending on build flags used, the PSE could be of type `pse` (which uses a 32-bit X86 compiler and toolchain), `pse_RV32` (which uses a 32-bit RISC-V compiler and toolchain) or `pse_RV64` (which uses a 64-bit RISC-V compiler and toolchain).

For a PSE of type `pse`, register `eax` is used when returning values and all parameters are passed on the stack, pointed to by register `esp`. The intercept library must get handles to these x86 registers and store them for future use:

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

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

7.1.2.2 ABI Intercepts

When using *ABI* intercepts, no knowledge of the PSE ABI is required.

7.2 Obtaining Intercepted Function Arguments

7.2.1.1 Raw Intercepts

When using *raw* intercepts, the peripheral model must explicitly access function arguments. To read arguments when using a PSE of type `pse`, all arguments are passed on the stack. The `esp` register containing the stack has already been stored by the constructor; standard code to read an argument from the intercepted function is as follows:

```
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);
}
```

7.2.1.2 ABI Intercepts

When using *ABI* intercepts, no knowledge of the PSE ABI is required (arguments are presented directly to the host intercept function).

7.3 Passing the Return Code from an Intercepted Function

7.3.1.1 Raw Intercepts

When using *raw* intercepts, the peripheral model must explicitly assign a function result to an appropriate register if required. For a PSE of type `pse`, a 32-bit function result is returned in `eax`. Before returning from the intercepted function the result is written into this register using `vmiosRegWrite()`.

```
Bool result = nativeFunction();

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

7.3.1.2 ABI Intercepts

When using *ABI* intercepts, no knowledge of the PSE ABI is required (the function result is returned directly from the host intercept function).

7.4 Data Exchange

This shows how data is transferred from the PSE’s memory at address `pseAddr` into native memory at `hostData`. `cachedRegion` provides a hint to the simulator, which can improve simulation performance if this routine is used often. `MEM_AA_FALSE` indicates that this is an artifact access.

```
char hostData[128];
memRegionP cache = 0;

void getData() {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);

    vmirtReadNByteDomain(
        domain,
        pseAddr,
        hostData,
        128,
        &cache,
        MEM_AA_FALSE
    );
}
```

8 Peripheral Model Example

This section describes an example of some of the peripheral modeling features introduced in the earlier sections.

The example is a DMA controller in a system with a RISC-V processor and two memory regions. This is a simplified model which does not represent a particular DMA device.

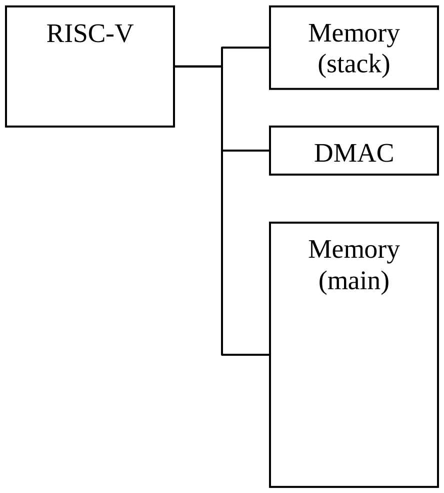


Figure 1: Example Virtual Platform Block Diagram

8.1 Example Source

`Directory: Examples/Models/Peripherals/creatingDMAC`

The examples are a progression of self-contained models, each building on the last. To see the full source code of each model, refer to to the directory above. The file `Examples/Models/Peripherals/creatingDMAC/INFORMATION.README.txt` describes the files and how to run the example.

8.2 IGEN

Each example uses the iGen productivity tool to generate most of the code. iGen is described in `iGen_Peripheral_Generator_User_Guide.doc`

To view the code (both generated and hand-written) each example should be run:
Copy the complete directory to your own working area then set up your environment to use Imperas Tools. In the new directory execute the script `example.sh` on Linux or `example.bat` on Windows.

8.2.1 Register model

Directory: `Examples/Models/Peripherals/creatingDMAC/1.registers`

The peripheral model files are in `peripheral/pse`. The model has its major parts but no behavior.

`peripheral/pse/pse.attrs.igen.c`
This contains the simulator interface structure `modelAttrs` that describes the model type, identification, and callbacks. Functions `nextParameter()` and `nextBusPort()` are iterators that return each parameter or bus descriptor in turn. Functions `peripheralSaveState()` and `peripheralRestoreState()` save or restore model state so that a session can be saved and then later restored. Note that the state of the model's PSE memory is automatically saved; only state outside this environment needs to be saved in these functions.

`peripheral/pse/pse.igen.h`
This declares storage types to represent register contents; each register is described by a union of its bitfields and its complete value. Other state variables and function prototypes are also declared here. The lines

```
#include "peripheral/bhm.h"
#include "peripheral/ppm.h"
```

include the PPM and BHM APIs.

`peripheral/pse/pse.igen.c`
This reserves space for the model's state. It contains the constructor which makes registers and ports. After calling the constructors, the main thread of the model waits until the end of simulation.

`peripheral/pse/dmac.user.c`
Originally created by iGen, this file contains templates for each function. It can be edited by hand, adding code to model the behavior of the peripheral. In this example the read and write function for each register checks if the access is a legal size then copies to or from the storage. In this model there is no behavior.

Application
The application code is in the `application` directory. It runs on a RISCV processor. It is written as if the model is complete; it commands the DMAC to transfer date from place to place, but of course, the DMAC does nothing.

8.2.2 Parallel Operations and Signaling Events

Directory: `2.parallelThreadsAndEvents`

`peripheral/pse/dmac.user.c`
This file has been expanded to model part of the DMA mechanism. The function `channelThread()` is a peripheral thread (see section 5.4) which waits for an event (see section 5.4.1) then runs one DMA operation (function `dmaBurst()` not complete).
The DMAC has two channels which run independently so has two threads, started in `userInit()` by `bhmCreateThread()`. An event is triggered in function

`writeAndStart()` when registers `ab32Ch0_config` or `ab32Ch1_config` are written. Once started, the thread should make the DMA transfer.

8.2.3 Master Memory Access

Directory: `3.memAccess`

`peripheral/pse/pse.igen.c`
The function `installMasterPorts()` opens address spaces for DMA reads and writes using `ppmOpenAddressSpace()`. The reads and writes will be on the buses that the bus ports are connected to in the platform.

`peripheral/pse/dmac.user.c`
The function `dmaBurst()` models DMA operation. Source,destination addresses, byte counts and transfer sizes are read from their registers. The main loop `while(bytes)` reads from simulated memory using `ppmReadAddressSpace()` and writes using `ppmWriteAddressSpace()`. A call to `bhmWaitDelay()` simulates time taken by the DMA operation.

Once started, the thread will make the DMA transfer without interruption until complete (this model does not cope with bus errors or exceptions).

8.2.4 Interrupts

Directory: `4.interrupt`

The DMAC notifies the processor when a DMA operation is complete by asserting an interrupt output.

`peripheral/pse/pse.igen.c`
The function `installNetPorts()` sets the net port handle `handles.INTTC` to the net connected in the platform to net port `INTTC`.

`peripheral/pse/dmac.user.c`
At the end of a DMA operation the function updateNet() writes a 1 to the interrupt net port. The port is returned to 0 when the appropriate registers are written.

8.2.5 Behavior using Native host code

Directory `5.nativeBehaviour`

This example illustrates the use of native host code in a peripheral model.
The example is contrived; there's no need to use native code in this case; but it shows how to build an intercept library and to call host functions from inside the PSE's environment. One of the two DMA channels uses native code to directly transfer data into the simulated memory. By convention, PSE code is in the directory `peripheral/pse`; host code in the intercept library is in `peripheral/model.`

`peripheral/pse/dmac.user.c`

Stub functions `initSemiHost` and `transferDataNative` will be intercepted by the simulator and transfer control to the intercept library. `initSemiHost` is called from the constructor. `transferDataNative` is called from within `dmaBurst.`

`peripheral/model/peripheral_native.c`

The lines

```
#include "vmi/vmiMessage.h"
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiOSLib.h"
#include "vmi/vmiPSE.h"
#include "vmi/vmiRt.h"
```

include the VMI API required in host code.

The structure `modelAttrs` is used by the simulator to locate the intercept library entry points. The `.intercepts` table defines the intercepted functions and their callbacks:

```
    .intercepts    =
    //   --------------------    ------- ---------------------------------------
    //   Name                     Opaque Callback
    //   --------------------     ------ ---------------------------------------
    {
        {"transferDataNative", 0, True,  VMIOS_ABI_INT('4', "po444", transferDataNative)},
        {"initSemiHost",       0, True,  VMIOS_ABI_INT('4', "poa4",  initSemiHost)      },
        {0}
    }
```

This example uses *ABI* intercepts: callbacks are defined using the `VMIOS_ABI_INT` macro. The first argument to the `VMIOS_ABI_INT` macro is a *function return type* character. This can be any of:

0:	void intercepted function
'b':	intercepted function returning `Boolean`
'4':	intercepted function returning `Uns32` or `Int32`
'8':	intercepted function returning `Uns64` or `Int64`
'a':	intercepted function returning `Addr` (a pointer in PSE application code)
'd':	intercepted function returning `double`

The second argument to the `VMIOS_ABI_INT` macro is a *parameter format string*, which describes the intercept function parameters. Characters in this string can be any of the ‘b’, ‘4’, ‘8’, ‘a’ or ‘d’ as described above, or additionally any of these *implicit argument* format specifiers:

'p':	the current processor (`vmiProcessorP`)
'o':	the current object (`vmiosObjectP`)
'c':	the name of the intercepted function

⇒ Note that any pointer parameter in PSE application code should be specified using the ‘a’ format specifier and handled as an `Addr` parameter in the intercepted function. Doing this will ensure that the intercept library will behave correctly irrespective of whether 32-bit or 64-bit PSE architecture is used.
⇒ Note that any *implicit* arguments (‘p’, ‘o’ or ‘c’) must precede all *explicit* arguments in the parameter list.

In this example, the `transferDataNative` has the following prototype to match the argument descriptions specified in the table:

```
static Uns32 transferDataNative(  // return type ‘4’
    vmiProcessorP processor,      // matches implicit ‘p’ parameter
    vmiosObjectP  object,         // matches implicit ‘o’ parameter
    Uns32         addressSrc,     // matches ‘4’ parameter
    Uns32         addressDest,    // matches ‘4’ parameter
    Uns32         bytes           // matches ‘4’ parameter
)
```

The matching function definition in the PSE application code is:

```
//
// Semihosted function: performs the DMA using native code for a configured channel
//
NOINLINE Uns32 transferDataNative(Uns32 src, Uns32 dest, Uns32 thisAccess)
{
    bhmMessage("F", PREFIX , "Failed to intercept %s", __FUNCTION__);
    return 0;
}
```

This function has an `Uns32` return code and three arguments (matching the last three explicit parameters specified for the host code `transferDataNative` function). *Take great care to ensure that the parameters of the PSE application function and host intercept function are consistent and correctly described by the `VMIOS_ABI_INT` macro parameters: mistakes here can be difficult to find.*

During initialization the intercept library finds the memory domain to read and write DMA data (they are the same in this example, but could be different).

```
object->portReadDomain = vmipsePlatformPortAttributes(
    processor,
    portReadName,
    &lo, &hi, &isMaster, &isDynamic
);
```

When a DMA transfer is required, this code reads and writes data to the simulated memory:

```
char tmp[MAX_BYTES];
vmirtReadNByteDomain (object->portReadDomain, addressSrc, tmp, bytes, 0, False);
vmirtWriteNByteDomain(object->portWriteDomain, addressDest,tmp, bytes, 0, False);
```

8.3 Platform Overview

The virtual platform is created using a module definition and executed using the harness program (harness.exe). The module is created using *iGen.*

This example uses a RISC-V processor model, the DMAC and generic OVP memory.

For a full description of virtual platform creation commands start with the user guides "iGen Platform and Module Creation User Guide" and "Writing Platforms and Modules in C User Guide".

8.3.1 Virtual Platform Design

This section describes the virtual platform.

8.3.1.1 Virtual Platform Memory Map

Figure 2 shows the memory map of the virtual platform.

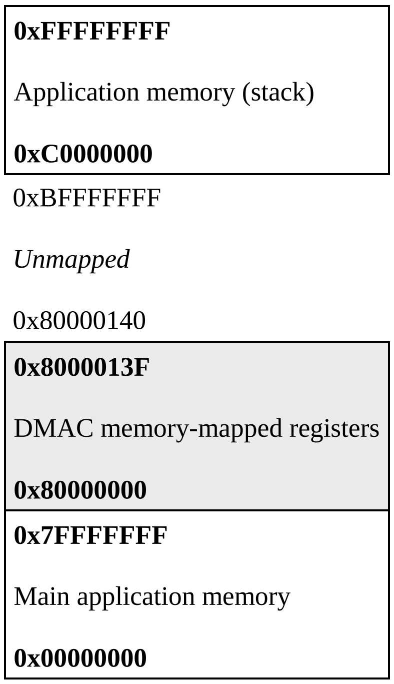


Figure 2: Memory map of platform

8.3.1.2 Virtual Platform Module Definition

Create the module:

```
ihwnew -name rv32WithDMACPeripheral
```

Add the bus to which the devices can be connected

```
ihwaddbus -instancename bus -addresswidth 32
```

Create the memories and connect to the bus. The base addresses of the memories and their sizes are defined when they are connected to the bus.

```
ihwaddmemory -instancename ram1 -type ram
ihwconnect   -instancename ram1 -busslaveport sp1 -bus bus \
                                -loaddress 0x00000000 -hiaddress 0x7fffffff

ihwaddmemory -instancename ram2 -type ram
ihwconnect   -instancename ram2 -busslaveport sp1 -bus bus \
                                -loaddress 0xc0000000 -hiaddress 0xffffffff
```

Add the processor and set its variant parameter:

```
ihwaddprocessor -instancename cpu1 \
                -vendor riscv.ovpworld.org -type riscv \
                -library processor -version 1.0 \
                 -semihostname pk \
                 -variant RV32I
```

Connect both processor ports to the bus:

```
ihwconnect -instancename cpu1 -busmasterport INSTRUCTION -bus bus
ihwconnect -instancename cpu1 -busmasterport DATA        -bus bus
```

Add the peripheral and connect it to the bus. The port name ‘DMACSP’ of the slave port must match the name in the peripheral model. The size of the port must match the sized in the call to `ppmOpenSlaveBusPort()`.

```
ihwaddperipheral -instancename dmac -modelfile peripheral/pse
ihwconnect       -instancename dmac -busslaveport DMACSP -bus bus \
                                    -loaddress 0x80000000 -hiaddress 0x8000013f
```

The module files generated by *iGen* are compiled to a host shared object using the provided `Makefile` and loaded by the `harness.exe` program to execute. `harness.exe` has a command line parser which allows *`–program`* to be used to load the application elf file into the RISC-V processor memory.

The simulation will run until it is interrupted or until the application finishes.

8.4 Peripheral Model Template

The peripheral model template is generated as C code with further user C code added to provide the definition of the behavior. It is compiled using a compiler toolchain to run on a PSE.
This section describes the TCL code used to create the C files in example directory `4.interrupt`. TCL in the other examples is similar.

TCL used by *iGen* for the peripheral model template creation is described in detail in the user guide “iGen Peripheral Generator User Guide”

First, create the peripheral:
`-name -vendor -library –version` sets its location in the component library.
`-constructor` and `-destructor` create stub functions to be completed.
`-formalvalues` generates code to fetch model parameters into variables of the same name as the formal parameter.
`-endianparam endian` adds a formal parameter of type `endian` and makes all memory mapped registers use the `endian` setting, which automatically byte-swaps data into and out of the registers.
`-nbyteregisters` chooses the new register interface. (Older models use a deprecated register interface limited to a maximum of 32 bits per register).

```
imodelnewperipheral  \
    -name        dmac \
    -vendor      ovpworld.org  \
    -library     peripheral \
    -version     1.0 \
    -constructor constructor \
    -destructor  destructor  \
    -nbyteregisters \
    -endianparam endian\
    -formalvalues
```

Define the slave port which will contain the memory-mapped registers.
`–mustbeconnected` generates code that raises an error if the port is not connected in the platform.

```

# Slave port for all control registers
imodeladdbusslaveport -name DMACSP -size 0x140 -mustbeconnected
```

Create an address block to group the 8-bit registers together.
NOTE: In this model the 8 bit registers are aligned onto a 4 byte boundary and there are gaps between some of the registers.

```
imodeladdaddressblock -port DMACSP -name ab8 -width 8  -offset 0 -size 0x40
```

Create the 8 bit registers:

```
imodeladdmmregister -addressblock DMACSP/ab8 -name intStatus    -offset 0x00 -
access r
imodeladdmmregister -addressblock DMACSP/ab8 -name intTCstatus  -offset 0x04 -
access rw -writefunction TCclearWr
imodeladdmmregister -addressblock DMACSP/ab8 -name intErrStatus -offset 0x0C -
access rw -writefunction errClearWr
imodeladdmmregister -addressblock DMACSP/ab8 -name rawTCstatus  -offset 0x14 -
access r
imodeladdmmregister -addressblock DMACSP/ab8 -name rawErrStatus -offset 0x18 -
access r
imodeladdmmregister -addressblock DMACSP/ab8 -name enbldChns    -offset 0x1C -
access r
imodeladdmmregister -addressblock DMACSP/ab8 -name config       -offset 0x30 -
access rw -writefunction configWr
```

Add the interrupt output port:

```
imodeladdnetport -name INTTC -type output
```

8.5 Peripheral Model Entry

This section describes the code created by *iGen*.

8.5.1 Attribute Table

The simulator obtains information from the peripheral model by examining the entries in the attribute table. The attribute table is of type `ppmModelAttrS` and must be called `modelAttrs`. It is in `pse.attrs.igen.c`.

```
ppmModelAttr modelAttrs = {

    .versionString    = PPM_VERSION_STRING,
    .type             = PPM_MT_PERIPHERAL,

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

    .saveCB        = peripheralSaveState,
    .restoreCB     = peripheralRestoreState,

    .vlnv          = {
        .vendor  = "ovpworld.org",
        .library = "peripheral",
        .name    = "dmac",
        .version = "1.0"
    },

    .family    = "ovpworld.org",

    .releaseStatus = PPM_UNSET,
    .visibility     = PPM_VISIBLE,
    .saveRestore    = 0,

};
```

8.5.2 The Main Function

`main()` is the model's entry point, called when platform construction is complete and before application processors are started. It is found in the generated file `pse.igen.c`.

`main()`adds some documentation then installs a callback function used to change the diagnostic level. This function sets a local variable which can be tested to control diagnostic output.

```
////////////////////////////////////// Main //////////////////////////////////////

int main(int argc, char *argv[]) {
    ppmDocNodeP Root1_node = ppmDocAddSection(0, "Root");
    {
        ppmDocNodeP doc2_node = ppmDocAddSection(Root1_node, "Description");
        ppmDocAddText(doc2_node, "DMAC peripheral model");
    }

    diagnosticLevel = 0;
    bhmInstallDiagCB(setDiagLevel);
    constructor();

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

```
Uns32 diagnosticLevel;

/////////////////////////// Diagnostic level callback //////////////////////////
static void setDiagLevel(Uns32 new) {
    diagnosticLevel = new;
}
```

`main()` then calls the constructor function, into which user construction code can be added and which calls the generated `periphConstructor()` function from which functions are called to add port and net connections.

```
////////////////////////////////// Constructor //////////////////////////////////

PPM_CONSTRUCTOR_CB(periphConstructor) {
    installSlavePorts();
    installRegisters();
    installMasterPorts();
    installNetPorts();
}
```

At the end of `main()` the peripheral waits for the end of simulation event.

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

At the end of simulation the `destructor()` function will be called. This can be used to report statistics. There is no need to free memory in the peripheral model.

8.5.3 Information about available ports

The bus, net, conn and packetnet ports are all defined and accessed by specific iteration functions registered in the attribute table.

The port iteration functions are in the generated file `pse.attrs.igen.c`

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

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

static ppmNetPort netPorts[] = {
    {
        .name            = "INTTC",
        .type            = PPM_OUTPUT_PORT,
        .mustBeConnected = 0,
        .description     = "Interrupt Request"
    },
    { 0 }
};

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

8.5.4 Information about parameters

The parameters for a peripheral model must be defined so that the simulator can obtain their names, types and any default, minimum and maximum values if appropriate for the type.

The definition of the iterator function to access the parameters is in the generated file `pse.attrs.igen.c`

```
static ppmParameter parameters[] = {
    {
        .name        = "endian",
        .type        = ppm_PT_STRING,
        .description = "Specify the endian of the processor interface",
    },
    { 0 }
};

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

8.5.5 Creating a Slave Port Interface

The code to create the slave port interface is in `pse.igen.c`

The peripheral slave port creates a window in the address space of the bus model to which it is connected. This window also appears in the address space of the PSE at the address returned by `ppmCreateSlaveBusPort()`

```
    handles.DMACSP = ppmCreateSlaveBusPort("DMACSP", 320);
```

The port name provides the link between the peripheral model and the platform.
If the port is connected to a bus model in the platform, any access to the window on the simulated bus will access the memory in the PSE. This memory is mapped to peripheral registers.

8.5.6 Registers

The code to create the memory mapped registers and install their callbacks is in `pse.igen.c`

8.5.6.1 Installing a Register

A register is created using `ppmCreateNByteRegister()`.

The function creates a register in the peripheral model. The register has a name and description and can be provided with a function used by the debugger to allow access without side effects. Additionally, read and write *view events* are constructed.

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.

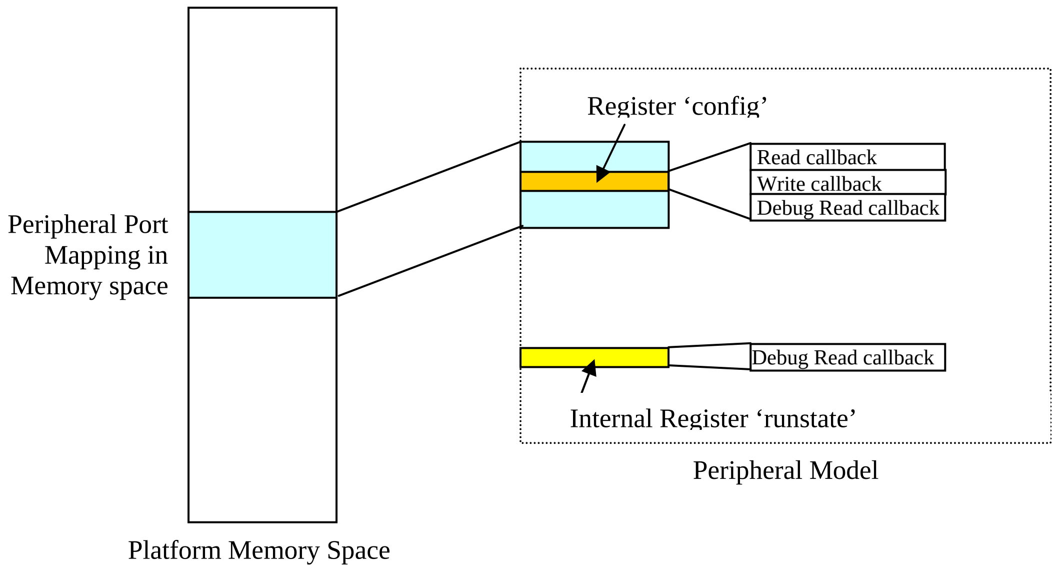


Figure 3: Example Peripheral Registers

`ppmCreateNByteRegister()` creates the register object and associates read and write callbacks that provide the behavior behind a register, and sets the register’s byte offset from the base address of the port.

```
Uns8 config;

    ppmCreateNByteRegister(
        "config",           // name
        "configuration",    // description
        DMACSP_Window,      // window base
        0x30,               // offset in bytes from the window’s base
        sizeof(config),     // register size in bytes
        configRd,           // read callback
        configWr,           // write callback
        configView,         // view callback
        &config,            // storage for this register
        0,                  // user data (not used)
        False,              // this register is not volatile,
        readable,           // If true, this register is readable
        writable,           // If true, this register is writable
        BHM_ENDIAN_LITTLE   // no byte swapping
    );
```

In the example a register 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` bytes from the base.

If reading the register has side effects (changes the peripheral's state), the read behavior of the register must be modeled in the function `configRd.` If there are no side effects the function can be omitted and the parameter set to zero.

If writing the register has side effects, the write behavior of the register must be modeled in the function `configWr.` If there are no side effects the function can be omitted and the parameter set to zero.

The variable `config` (which must be the correct size for the register) holds the state of the register. If the read or write functions are omitted this variable will be read or updated automatically.

If the `configRd` function is supplied and the true value of the register is not stored in the config variable, then the function `configView` must be supplied. The debugger will use this to read the register so the function it must not change the peripheral's state. As an example, a true read of the data register of a serial device will be destructive; it will cause the next data item to be available. Viewing the value of the data register in the debugger should not destroy the data.

If the `configRd` is not supplied, then the `readable` parameter controls if a read access is allowed. If the `configWr` is not supplied, then the `writable` parameter controls if a write access is allowed. If a register is neither readable nor writable then it will not appear in the peripheral's memory map so `ppmCreateNByteInternalRegister()`should be used instead.

`ppmCreateNByteInternalRegister()` creates a register that is visible to the debugger but is not memory-mapped.

```
Uns8 runstate;

    ppmCreateNByteInternalRegister(
        "runstate",                   // name
        "operational status",         // description
        sizeof(runstate),             // register size in bytes
        0,                            // debug view (not used)
        &runstate                     // storage for this register
        0                             // user data (not used)
    );
```

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. Its value is stored in the `runstate` variable.

Arrays of similar registers can be modeled without duplicating the callbacks; the `userdata` field for each register is supplied with a different value. All similar registers are supplied with the same callbacks. Each callback receives the `userData` field and uses this to distinguish which register was accessed. The `userData` value could be an integer offset into an array of registers or a pointer to a structure representing one register.

8.5.7 Running the Example

This section describes 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 `Examples/Models/Peripherals/creatingDMAC/1.registers` are scripts `example.sh` and `example.bat` that will build the the module, the test application and generate and the peripheral PSE template and the user behavioral code.

The script will perform the commands to build:

```
bash> make -C application
bash> make -C module          NOVLNV=1
bash> make -C peripheral/pse   NOVLNV=1
```

`harness.exe` loads the module and runs the simulation:

```
bash> harness.exe \
         --modulefile module/model \
         --program application/dmaTest.RISCV32.elf
```

Output should be similar to this:

```
OVPsim (32-Bit) v20160627.0 Open Virtual Platform simulator from
www.OVPworld.org.
Copyright (c) 2005-2016 Imperas Software Ltd.  Contains Imperas Proprietary
Information.
Licensed Software, All Rights Reserved.
Visit www.IMPERAS.com for multicore debug, verification and analysis solutions.

OVPsim started: Wed Sep 07 17:25:59 2016


TEST DMA: dmaBurst ch:0  bytes:13
TEST DMA: dmaBurst ch:1  bytes:35
TEST DMA: DMAC Register Read DMA_C0_SRC_ADDR      0xffffefe4
TEST DMA: DMAC Register Read DMA_C0_DST_ADDR      0xffffdfe4
TEST DMA: DMAC Register Read DMA_C0_CONTROL       0x0000000d
TEST DMA: DMAC Register Read DMA_C0_CONFIGURATION 0x00000001
TEST DMA: DMAC Register Read DMA_C1_SRC_ADDR      0xffffcfe4
TEST DMA: DMAC Register Read DMA_C1_DST_ADDR      0xffffbfe4
TEST DMA: DMAC Register Read DMA_C1_CONTROL       0x00000023
TEST DMA: DMAC Register Read DMA_C1_CONFIGURATION 0x00000001

OVPsim finished: Wed Sep 07 17:26:01 2016
```

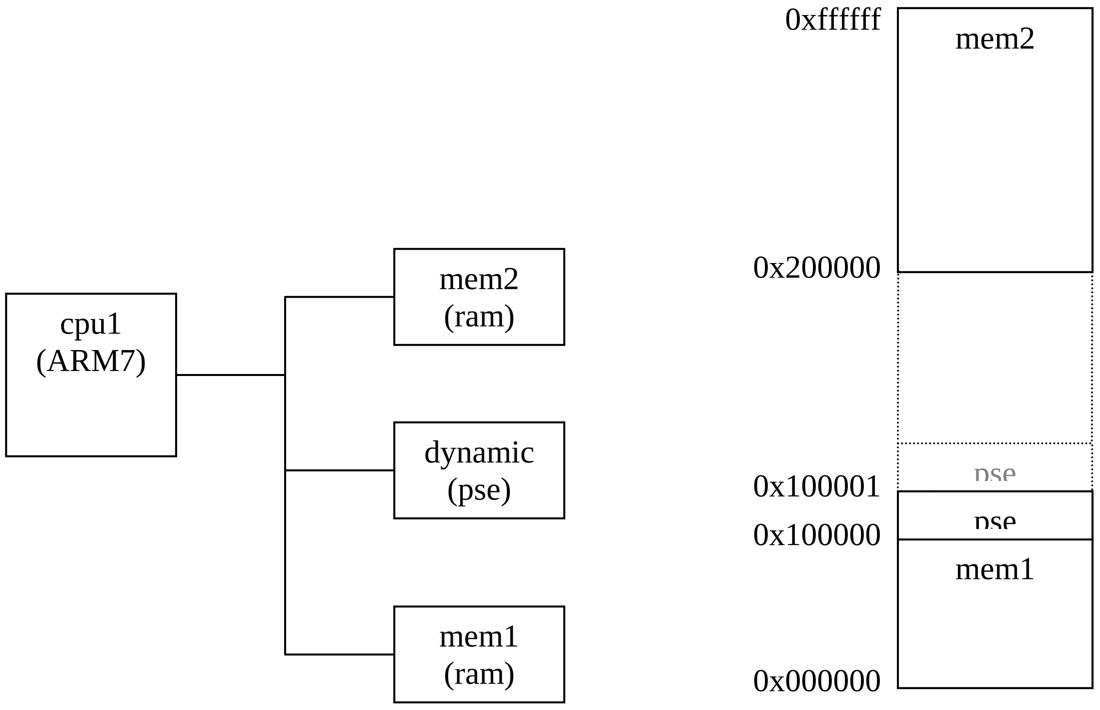
9 Dynamic Slave Port example

Directory: `$IMPERAS_HOME/Examples/Models/Peripherals/dynamicSlavePort`

The PCI bus protocol lets a bus device set the address of its slave port itself. This example that shows how a peripheral model can have this behaviour.

The example is in three parts, an application, a module hardware definition and a peripheral with a dynamic slave port.

The module assembles the components shown on the left. The memory map is on the right. The dynamic slave port can be mapped anywhere in the unmapped region.



The peripheral has one register accessible through its slave port. The *remap* register is at offset zero. When read it provides the current address mapping. When written it moves the base address of the dynamic slave port.

Take a copy of the example directory and compile the test application, module and peripheral using the following commands:

```
cp -r $IMPERAS_HOME/Examples/Models/Peripherals/dynamicSlavePort .
cd dynamicSlavePort
make all
```

9.1 Instance Peripheral in Module

In `module/module.op.tcl` the peripheral is instanced in the normal way but instead of defining the low and high address for the slave port connection, no address information is provided:

```
ihwaddperipheral -instancename dynamic -modelfile peripheral/pse.pse
ihwconnect       -instancename dynamic -bus bus -busslaveport hostif
```

There is also no size information provided when the peripheral is instanced so it is both dynamically located and sized in the address map.

9.2 Peripheral iGen Definition

The peripheral interface is defined in the iGen file `peripheral/pse.tcl`, as shown below.

The only difference in the definition between a static and a dynamic port is the use of the additional `--remappable` argument:

```
set slvPrt "hostif"
set adrBlk "ab"
set size    4

# Dynamic slave port connection
imodeladdbusslaveport -name $slvPrt -size $size -mustbeconnected -remappable
```

The address block and register are added into the peripheral model in the same way for a dynamic or a static mapping:

```
# Address block
imodeladdaddressblock -name $adrBlk -port $slvPrt \
     -width 32 -offset 0x0 -size $size

# Registers
imodeladdmmregister -name remap  -addressblock $slvPrt/$adrBlk \
    -offset 0x0  -access rw -writefunction writeRemap
```

9.3 Peripheral User Code

9.3.1 Initialization

File: `peripheral/pse.user.c`

In `constructor()` the handle to the port `hostif` is initialized to a region the size of the slave port region. The generated constructor is called, then with the `remap` register set to its reset value, the initial mapping is made.

```
static void portMap(void) {
    ppmCreateDynamicSlavePort(
        "hostif",
        hostif_ab_data.remap.value,
        PORT_SIZE,
        handles.hostif
    );
}

PPM_NBYTE_WRITE_CB(writeRemap) {

    bhmMessage("I", "MAP", "Old Mapping 0x%0x", hostif_ab_data.remap.value);

    // Delete previous mapping
    ppmDeleteDynamicSlavePort("hostif", hostif_ab_data.remap.value, PORT_SIZE);


    hostif_ab_data.remap.value = *(Uns32*)data;

    bhmMessage("I", "MAP", "New Mapping 0x%0x", hostif_ab_data.remap.value);

    portMap();
}

PPM_CONSTRUCTOR_CB(constructor) {

    handles.hostif = malloc(PORT_SIZE);

    periphConstructor();

    portMap();
}
```

9.3.2 Dynamic mapping

The function `writeRemap()` removes the old mapping then installs the new, using the same function as `constructor()`.

9.4 Running the Example

To run the simulation, in the `dynamicSlavePort` directory, run:

```
harness.exe \
    --modulefile module/model.${IMPERAS_SHRSUF} \
    --program application/application.ARM7.elf
```

You should see the following output as the processor executes the application and accesses the peripheral registers:

```
APP: Starting ..
APP: Read Re-Map Reg 0x10000000
APP: Write Re-Map Reg : Move to 0x10000100
Info (MAP) testDynamic/dynamic: Old Mapping 0x10000000
Info (MAP) testDynamic/dynamic: New Mapping 0x10000100
APP: Read Re-Map Reg 0x10000100
```

The application attempts to read the old mapping. There is nothing there and because the processor is not programmed to simulate exceptions, the simulation stops with a processor exception.

```
APP: Attempt to access old mapping at 0x10000000 - expect failure
Processor Exception (PC_PRX) Processor 'testDynamic/cpu1' <ADDRESS>: ldr etc
Processor Exception (PC_RPX) No read access at 0x10000000
```

10 QuantumLeap with Peripherals

Imperas Professional Simulation products implement a parallel simulation algorithm called *QuantumLeap,* which enables platform simulation to be distributed over separate threads on multiple cores of the host for improved performance. This section shows how to enable the QuantumLeap algorithm for peripherals and describes how simulation performance and results are affected.

QuantumLeap allows for the parallel execution of both processor and peripheral models on host processors. This section describes its use with peripherals; refer to the *OVPsim and CpuManager User Guide* for information about parallelization of processors.

10.1 The QuantumLeap Algorithm

The QuantumLeap peripheral algorithm allows peripheral model components implemented as native code on the host processor to be run in a thread in parallel with other peripheral models and also the main simulation thread. To use the algorithm, key parts of peripheral behavior must be implemented using *intercepted functions* (see section 7).

The QuantumLeap Peripheral algorithm is suitable for use in situations where a single peripheral function is compute-intensive. An example might be a peripheral that implements a complex encryption algorithm.

When QuantumLeap is enabled, the simulation flow is as follows:

1. An intercepted function in the peripheral is launched in a separate native thread.
2. Immediately after calling the intercepted function, the PSE issues a wait, either for a *fixed delay,* or for an unspecified *quantum delay*.
3. The peripheral and main simulation threads are then run in parallel.
4. If a *fixed delay* was specified when the parallel thread was launched, the simulator will resynchronize with the peripheral thread when the thread returns at that fixed simulation time. In other words, if the simulator thread reaches the specified simulated time before the peripheral thread completes, the main simulation thread will be suspended until the peripheral thread returns before continuing, and if the peripheral thread completes before the simulation thread reaches the specified time, then the main simulation thread will continue uninterrupted.
5. If a *quantum delay* was specified when the parallel thread was launched, the simulator will resynchronize with the peripheral thread at the next quantum boundary after the peripheral thread completes, however long that might take. In this case, the main simulation thread is never suspended.

Fixed delays are used when the purpose of the simulation is to model a specific timing aspect of peripheral behavior. For example, the peripheral might be implementing an encryption algorithm that is known to take 100us to complete in the real hardware. Using fixed delays, the simulation is deterministic.

Quantum delays are used when the purpose is simply to maximize simulation performance and there is no need to model specific real-world timing. The point at which the simulation thread and the peripheral thread resynchronize will vary from run to run depending on host load and other factors. The simulation will not be deterministic.

The peripheral QuantumLeap algorithm operates independently to the processor QuantumLeap algorithm – both or either can be enabled independently. Bear in mind that the two algorithms will, however, compete for limited host resources: it may be necessary to adjust the number of threads available for parallel processor simulation (see documentation of option `-parallelthreads` in the *OVPsim and CpuManager User Guide*) or reduce the number of peripherals run as separate threads for best performance.

10.2 Example

Directory: `Examples/Models/Peripherals/usingNativeThreading`

The example is in three parts; an application, a platform and a peripheral with an algorithm implemented as native code, in this case a *sort* algorithm. The peripheral performs a sort on an area of memory shared with the host so that the sort can be run as native code in a separate thread.

The number of peripherals that is instanced in the platform may be defined by setting the `PERIPHERAL` build variable. This must be set the same in the application and the platform and should be at least one less than the number of host processors. This allows another host processor for simulation of a processor in the platform that is controlling the peripheral execution (and itself executing a Dhrystones benchmark algorithm).

Compile the test application, platform and peripheral using the following commands in the `usingNativeThreading` directory:

```
export PERIPHERALS=2
make –C application
make –C module            NOVLNV=1
make –C peripheral/pse    NOVLNV=1
make –C peripheral/model  NOVLNV=1
```

10.2.1 Peripheral Code

File: `dataSort.user.c`

```
// Thread for each channel
static void channelThread(void *user) {

    for (;;) {
        if (DIAG_HI) bhmMessage("I", PREFIX, "Waiting\n");

        bhmWaitEvent(state.ch.start);

       if (DIAG_HI) bhmMessage("I", PREFIX, "Started\n");

        // run native sort algorithm
        runSort();
        // wait for simulated time of 30 seconds, or for as long as required
        // for asynchronous threads to run
        bhmWaitDelay(state.quantumDelay ? QUANTUM_DELAY : 30000000);


       if (DIAG_HI) bhmMessage("I", PREFIX, "Done\n");
        state.ch.dataReady = True;
        updateInterrupt();
    }
}
```

In the example, a sort algorithm is implemented in native code. This algorithm is executed when function `runSort()` is intercepted:

```
        runSort();
```

Following the intercepted sort function, the call to `bhmWaitDelay()`, either waits for a fixed time (30 seconds) or until the threaded sort function completes. The type of delay is set by parameter `quantumDelay`.

```
        bhmWaitDelay(state.quantumDelay ? QUANTUM_DELAY : 30000000);
```

Note that the special value `QUANTUM_DELAY` is used to indicate that the thread should wait until the quantum boundary after the native thread completes.

In threaded operation, the native function implementing `runSort` is launched in a thread. The PSE code continues immediately, executing the `bhmWaitDelay`. The PSE thread will then wait, either for the fixed 30 second delay, or until the quantum boundary after the native thread completes.

10.2.2 Peripheral Native Code

File: peripheral_semihost.c

```
    .intercepts    =
    //   --------------- ----------- ----------------------- ------------
    //   Name            Address     Attributes              Callback
    //   --------------- ----------- ----------------------- ------------
    {
        {"initSemiHost", 0,          OSIA_OPAQUE,            VMIOS_ABI_INT(0,
"o44b", initSemiHost)},
        {"runSort",      0,          OSIA_OPAQUE|OSIA_THREAD, runSort     },
        {0}
    }
```

QuantumLeap is enabled for a native intercepted function by setting bit field attribute `OSIA_THREAD` in the attribute table of the peripheral model native code. In this example, function `runSort` is specified to be capable of being run in parallel.

⇒ Note that threaded functions must be *raw* intercepts: *ABI* threaded intercepts are not supported. However, the `initSemiHost` intercept, which is not threaded, can be implemented as an ABI intercept. See section 10.4 for more information.

10.2.3 Module Definition

`File: module.op.tcl`

The example uses `harness.exe` to load a module.

The hardware definition creates a module then instances and connects the components.

```
ihwnew -name peripheralNativeThreadTest -vendor ovpworld.org -library module -
version 1.0

ihwaddbus -instancename bus -addresswidth 32

ihwaddnet -instancename int0

ihwaddprocessor -instancename cpu1 \
                -type arm \
                -vendor arm.ovpworld.org \
                -semihostname armNewlib \
                -semihostvendor arm.ovpworld.org \
                -endian little

ihwsetparameter -handle cpu1 -name variant -value Cortex-A9UP -type string

ihwconnect -instancename cpu1 -busmasterport INSTRUCTION -bus bus
ihwconnect -instancename cpu1 -busmasterport DATA        -bus bus

ihwconnect -instancename cpu1 -netport fiq -net int0

ihwaddmemory -instancename mem1 -type ram
ihwconnect   -instancename mem1 -busslaveport sp1 -bus bus \
             -loaddress 0x00000000 \
             -hiaddress 0x3fffffff

ihwaddmemory -instancename mem2 -type ram
ihwconnect   -instancename mem2 -busslaveport sp1 -bus bus \
             -loaddress 0xc0000000 \
             -hiaddress 0xffffffff
```

The instantiation of the number of peripheral models defaults to 2 but can be changed by specifying the environment variable PERIPHERALS when the module is built. This will result in the creation of a module with a specific number of peripherals at a base address `DATASORTSPBASE` and with a stride between them of `DATASORTSPSIZE` both of which can be modified with the appropriate environment variable setting.

```
#
# peripherals
#
set percount  2
set spbase    0x80000000
set spstride  0x1000
set spsize    0x013f
```

```
if { [ info exists env(PERIPHERALS) ]} {
    set percount $env(PERIPHERALS)
    puts "MODULE: Set Peripheral Count to $percount"
}
if { [ info exists env(DATASORTSPBASE) ]} {
    set spbase $env(DATASORTSPBASE)
    puts "MODULE: Set Peripheral Slave Port Base to 0x[format %08x $spbase]"
}
if { [ info exists env(DATASORTSPSIZE) ]} {
    set spstride $env(DATASORTSPSIZE)
    puts "MODULE: Set Peripheral Slave Port Stride to 0x[format %04x $spstride]"
}

ihwaddformalparameter -name registerOnly -type bool
ihwaddformalparameter -name quantumDelay -type bool

proc addPeripheral {id} {
    global spbase
    global spsize
    global spstride
    set abase [expr $spbase + ($spstride * $id)]
    set atop  [expr $abase  +  $spsize]
    # instance peripheral
    ihwaddperipheral -instancename sort${id} -modelfile peripheral/pse/pse.pse
    # connect slave and master ports to bus
    ihwconnect -instancename sort${id} -busslaveport  DATASORTSP -bus bus \
        -loaddress $abase \
        -hiaddress $atop

    ihwconnect -instancename sort${id} -busmasterport DATASORTMP -bus bus

    ihwconnect -instancename sort${id} -netport INT -net int0

    ihwsetparameter -handle sort${id} \
                    -name registerOnly \
                    -expression registerOnly -type bool

    ihwsetparameter -handle sort${id} \
                    -name quantumDelay \
                    -expression quantumDelay -type bool
}

# Add the peripherals
for {set i 0} {$i < $percount} {incr i} {
    addPeripheral $i
}
```

The peripheral model has parameters are `registerOnly` and `quantumDelay` which can be used to configure the behavior.

10.2.4 Threaded Operation

To run the simulation, showing threaded peripheral operation, in the `usingNativeThreading` directory, run:

```
harness.exe \
    --modulefile module/model.${IMPERAS_SHRSUF} \
    --program application/dataSortTest.ARM7.elf \
    --parallelperipherals
```

On the simulator command line *–parallelperipherals* enables the QuantumLeap algorithm.

You should see the following output as the processor executes the application and starts the peripheral data processing. *CPU1* is executing the Dhrystones benchmark while the peripherals `sort0` and `sort1`, are sorting data sets in the shared memory:

```
CpuManagerMulti (64-Bit) v20150901.0 Open Virtual Platform simulator from
www.IMPERAS.com.

...

Info (DATASORT) platform/sort0: Constructor called
Info (PP_CRT) PSE platform/sort0: creating thread 'datasortThread'
Info (DATASORT) platform/sort0: Waiting
Info (DATASORT) platform/sort1: Constructor called
Info (PP_CRT) PSE platform/sort1: creating thread 'datasortThread'
Info (DATASORT) platform/sort1: Waiting
...
Info (DATASORT) platform/sort1: Started
Info (DATASORT_SEMI) platform/sort1, reseeding peripheral data buffer

Dhrystone Benchmark, Version 2.1 (Language: C)

Info (DATASORT_SEMI) platform/sort0: runSort: sort 0x500000 words at 0x2fe68
Program compiled without 'register' attribute

Please give the number of runs through the benchmark:
Execution starts, 2000000 runs through Dhrystone
Info (DATASORT_SEMI) platform/sort1: runSort: sort 0x500000 words at 0x502fe68
Info (DATASORT) platform/sort0: Done
Info (DATASORT) platform/sort0: Interrupt signal asserted
Info (DATASORT) platform/sort0: Waiting
Info (DATASORT) platform/sort1: Done
Info (DATASORT) platform/sort1: Interrupt signal asserted
Info (DATASORT) platform/sort1: Waiting
TEST DATASORT: FIQ Interrupt
TEST DATASORT: check peripheral sort0
Info (PP_RDR) PSE platform/sort0: read register 'ab8_start' = 0x01
TEST DATASORT: active peripheral sort0
sorted[0] = 0x48d
sorted[524288] = 0x199a98d1
...
Bool_Glob:            1
        should be:   1
Ch_1_Glob:            A
        should be:   A
Ch_2_Glob:            B
        should be:   B
Arr_1_Glob[8]:        7
        should be:   7
Arr_2_Glob[8][7]:     2000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
        should be:   (implementation-dependent)
  Discr:              0
        should be:   0
  Enum_Comp:          2
        should be:   2
  Int_Comp:           17
...
```

```
TEST DATASORT: 6 interrupts received
Info
Info ---------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    0.03 seconds: PSE THREAD 'platform/sort1'
Info    0.02 seconds: PSE THREAD 'platform/sort0'
Info    0.06 seconds: PSE 'platform/sort1' (and 26 terminated callbacks)
Info    0.06 seconds: PSE 'platform/sort0' (and 26 terminated callbacks)
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info CPU 'platform/CPU1' STATISTICS
Info   Type                  : arm (Cortex-A9UP)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x22268
Info   Simulated instructions: 10,914,295,106
Info   Simulated MIPS        : 919.1
Info ---------------------------------------------------
Info
Info ---------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 109.14 seconds
Info   User time             : 19.28 seconds
Info   System time           : 0.02 seconds
Info   Elapsed time          : 11.88 seconds
Info   Real time ratio       : 9.19x faster
Info ---------------------------------------------------

CpuManagerMulti ParallelPeripheral finished: Tue Oct 20 16:25:35 2015
```

At the end of simulation if parallel threaded peripherals have operated, the word *ParallelPeripheral* will be seen in the end banner.

10.2.5 Non-Threaded Operation

To run the simulation with non-threaded peripheral operation run:

```
harness.exe \
    --modulefile module/model.${IMPERAS_SHRSUF}
```

You should see similar output to the threaded operation, however some of the ordering may be different as the peripherals are no longer executing in separate threads:

```
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    2000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
...
TEST DATASORT: 6 interrupts received
Info
```

```
Info --------------------------------------------------
Info PSE SIMULATION TIME STATISTICS
Info    3.70 seconds: PSE THREAD 'platform/sort1'
Info    3.71 seconds: PSE THREAD 'platform/sort0'
Info    0.02 seconds: PSE 'platform/sort1' (and 26 terminated callbacks)
Info    0.03 seconds: PSE 'platform/sort0' (and 26 terminated callbacks)
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info CPU 'platform/CPU1' STATISTICS
Info   Type                  : arm (Cortex-A9UP)
Info   Nominal MIPS          : 100
Info   Final program counter : 0x22268
Info   Simulated instructions: 10,914,295,106
Info   Simulated MIPS        : 602.2
Info --------------------------------------------------
Info
Info --------------------------------------------------
Info SIMULATION TIME STATISTICS
Info   Simulated time        : 109.14 seconds
Info   User time             : 18.09 seconds
Info   System time           : 0.03 seconds
Info   Elapsed time          : 18.22 seconds
Info   Real time ratio       : 5.99x faster
Info --------------------------------------------------

CpuManagerMulti finished: Wed Oct 21 09:06:24 2015
```

Compared with the previous results, PSE threads `platform/sort0` and `platform/sort1` are now taking significant time in the main simulation thread. Previously, times for these threads were very low, because most of the time was consumed by separate native threads, not the main simulation thread.

10.3 QuantumLeap Results

In this example, when threaded operation is enabled the simulation duration is about 12 seconds, allowing the ARM processor to run at 919 MIPS. When non-threaded operation is enabled, the simulation duration is about 18 seconds, allowing the ARM processor to run at about 600 MIPS. The difference is explained by the fact that in the second simulation the sort algorithms are executed in the main simulation thread.

The performance reported may vary and depends on the performance of the host and also the interaction of the processor applications with the peripheral operations. This example was run on a 2.4 GHz Dell Core i7-4700MQ.

10.4 Configuration of Peripheral Native Code

When QuantumLeap is enabled, peripheral native code will run in parallel to the invoking PSE. This means that if a native thread attempts to extract information from the PSE (for example, arguments on the processor stack or register values) it may no longer be valid, because the invoking PSE will have continued to execute.

Intercepted functions that run in parallel threads *should therefore never pass any arguments*: any initialization information should be passed using a previous initialization function that does not have threading enabled.

In this example the initialization is carried out using a separate function, `initSemiHost`, which is defined as an ABI non-threaded intercept in the interception attribute table:

```
    .intercepts    =
    //   --------------- ----------- ------------------------ ------------
    //   Name            Address     Attributes               Callback
    //   --------------- ----------- ------------------------ ------------
    {
        {"initSemiHost", 0,          OSIA_OPAQUE,             VMIOS_ABI_INT(0,
"o44b", initSemiHost)},
...
}
```

A function in the PSE application code is used to pass configuration data regarding the data buffer address and size:

```
//
// Initialize semihost if not already done
//
static void initSH(void) {
    if(!state.initSH) {
        state.initSH = 1;
        initSemiHost(
            DATASORTSP_ab32ch0_data.srcAddr.value,
            DATASORTSP_ab32ch0_data.dataSize.value,
            state.registerOnly
        );
    }
}
```

In the native code the arguments are saved:

```
static void initSemiHost(
    vmiosObjectP object,
    Uns32        base,
    Uns32        size,
    Bool         registerOnly
) {
    object->registerOnly = registerOnly;
    object->memoryBase   = base;
    object->memorySize   = size;

    ...
}
```

The threaded callback then uses the saved values to control its operation:

```
static VMIOS_INTERCEPT_FN(runSort) {

    // don't do operation if register interface only
    if (object->registerOnly) {

        ...

    } else {

        Uns32 base   = object->memoryBase;
        Uns32 size   = object->memorySize;
        ...
    }
}
```

11 Building Peripherals

11.1 OVP Library Structure

When building your own peripherals, it is recommended that you use a file structure identical to that in ImperasLib, and that you put your models in `ImperasLib/source/<your company URL>` or other unique name. This will ensure that the supplied `Makefile` can be used and that the simulator will be able to locate your models.

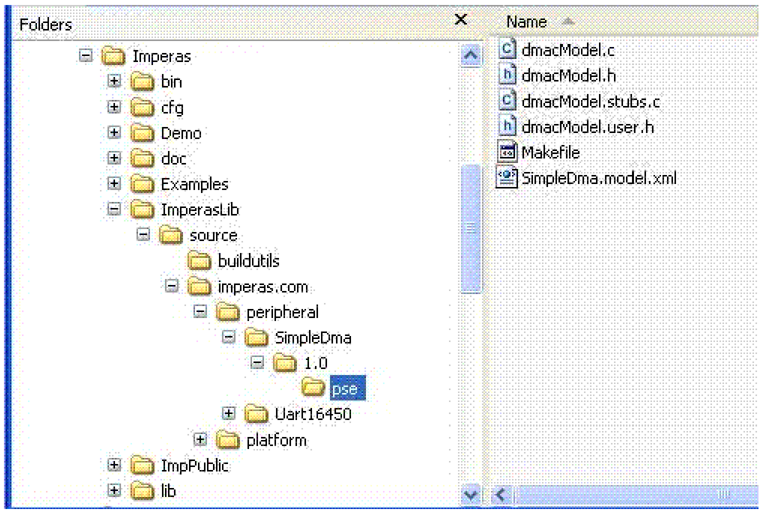


Figure 4: OVP Library Structure

11.2 Peripheral Model Makefiles

In this section we discuss building a peripheral.

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

- `ImperasLib/source`,
- `ImperasLib/source/imperas.com/peripheral/SimpleDma/1.0/pse`
- `ImperasLib/source/imperas.com/peripheral/SimpleDma/1.0/model`[3]

There are a set of `Makefiles` in `ImperasLib/buildutils` that are used to build different component types in the source library; for the peripheral model this is `Makefile.pse`.

The Makefile is includes in the component Makefile which will typically be:

```
ifndef IMPERAS_HOME
  IMPERAS_ERROR := $(error "IMPERAS_HOME not defined, please setup Imperas/OVP
environment")
```

[3] 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.

```
endif
IMPERAS_HOME := $(shell getpath.exe "$(IMPERAS_HOME)")

include $(IMPERAS_HOME)/ImperasLib/buildutils/Makefile.pse
```

11.2.1 Selecting PSE Type

The different types of Peripheral Simulation Engine, that may be used to implement the behavior defined in source code, is selected when the model is built by specifying `IMPERAS_PSE` in the Makefile in the pse directory.

IMPERAS_PSE Setting	PSE Implementation
PSE	32-bit x86 architecture (default)
PSE_RV32	32-bit RISC-V architecture
PSE_RV64	64-bit RISC-V architecture

For example, to build a PSE file to execute on a 64-bit RISC-V architecture PSE the following would be used in the Makefile.

```
…
IMPERAS_PSE=PSE_RV64
include $(IMPERAS_HOME)/ImperasLib/buildutils/Makefile.pse
```

The default PSE type (32-bit x86 architecture) requires use of a legacy GNU tool chain; this has a known problem that printing of 64-bit types (e.g. `Uns64` or `Addr`) is not supported by the `printf` implementation, which can make debugging PSE code difficult. As an alternative, either 32-bit or 64-bit RISC-V architecture can instead be used, both of which use a more modern GNU tool chain that does not have this problem.

Which PSE architecture to choose can be affected by two other factors:

1. If the PSE is being used to run pre-existing software, it is sometimes the case that the software makes implicit assumptions about the size of a pointer (4-byte or 8-byte). If this is the case, select a PSE architecture compatible with the software.
2. If the PSE requires to share complex data structures with an application processor, select a PSE for which the layout of structures in memory matches the application processor.

The simulator uses information encoded in the generated PSE executable to determine what architecture PSE to run, so this does not need to be explicitly specified. A single platform can use PSEs compiled with any combination of architecture if required.

11.3 Building Peripheral Models

11.3.1 Building to the Default Output Location

The default location for the output when building a library is the `$SYSTEMVLNV` location, which will be `$IMPERAS_HOME/lib/$IMPERAS_ARCH/ImperasLib`

To build into the default directory, in a shell (MSYS on Windows) type

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

11.3.2 Building to a Defined Output Location

To build into a specified directory, for example a local directory, for example `myLocalLib`, in a shell (MSYS on Windows) type

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

12 Troubleshooting

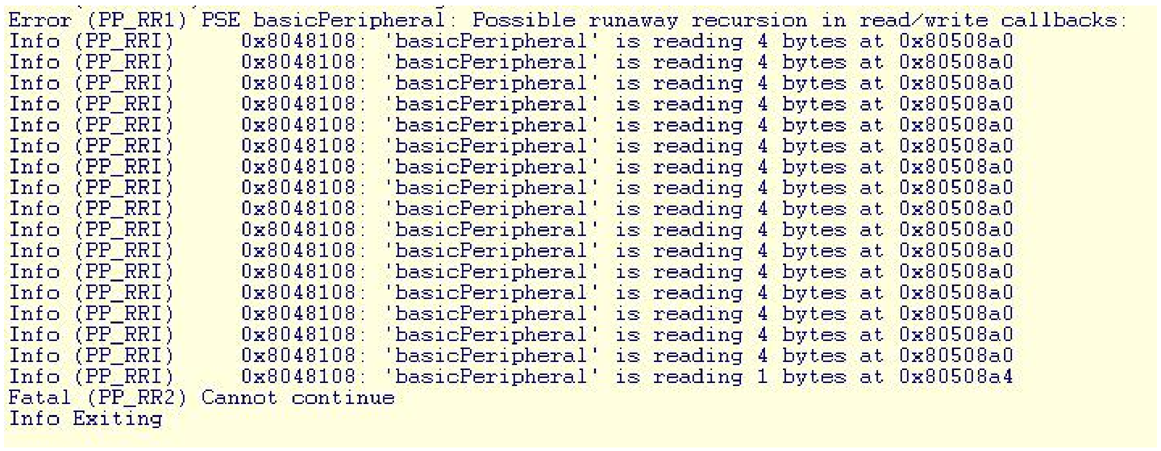
Here are some common problems encountered in peripheral models.

12.1 Runaway Recursion

12.1.1 Error Description

A region in PSE address space can have a callback installed. If code in this callback accesses the same addresses the simulator can detect the recursion. However if the code in the callback is intercepted, and if the host code in the intercept library uses a vmi function to access the same space, the simulator cannot detect the recursion.

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



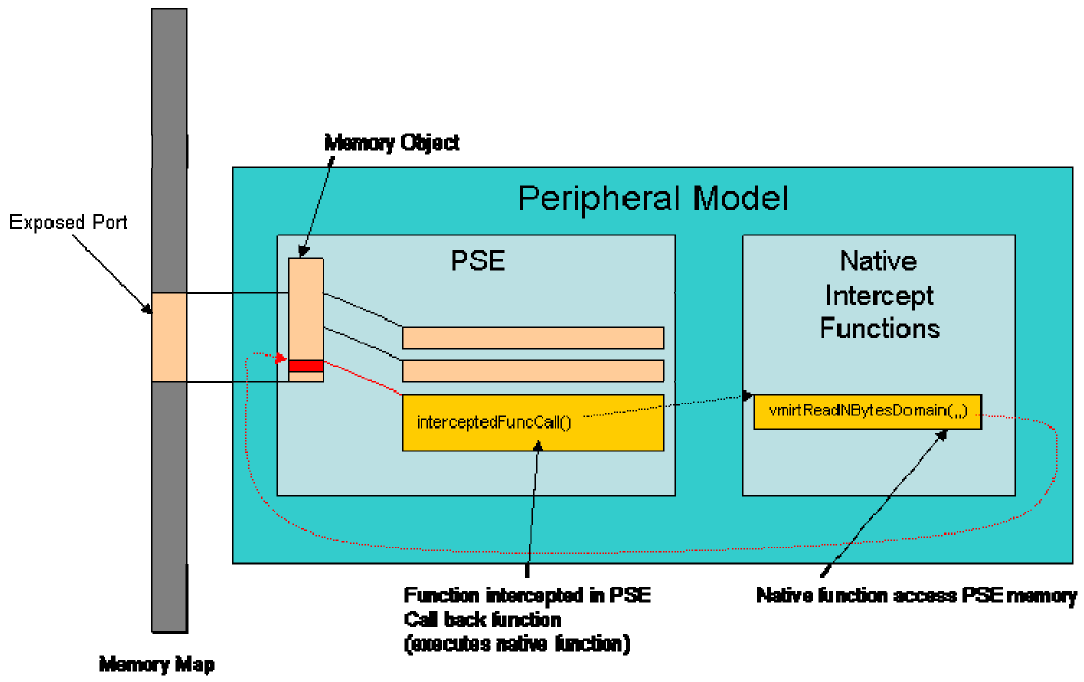
```
Error (PP_RR1) PSE basicPeripheral: Possible runaway recursion in read/write callbacks:
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 4 bytes at 0x80508a0
Info (PP_RRI)     0x8048108: 'basicPeripheral' is reading 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.

12.1.2 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.


Memory Object
Exposed Port
Peripheral Model
PSE
Native Intercept Functions
interceptedFuncCall()
vmirtReadNBytesDomain(,,)
Function intercepted in PSE Call back function (executes native function)
Native function access PSE memory
Memory Map

The following is an example of code that could cause this recursion 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 `window` the call back `readPort` is called. This function contains a call to an intercepted function. The intercepted function transfers control to the intercept library.

```
PPM_READ_CB(readPort){

    Uns32 val;

    semiReadData(&val);

    return val;
}
```

The function in the intercept library `semiReadData()` uses a VMI API function to read from an address which coincides with the peripheral port.

```
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);

}
```

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 to the same address that caused the original callback. It is going to call the callback again and cause the runaway recursion.

##