OVP BHM and PPM API Function Reference

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

Author:	Imperas Software Limited
Version:	2.11
Filename:	OVP_BHM_PPM_Function_Reference.doc
Last Saved:	Tuesday, 06 April 2021

Copyright Notice

Table of Contents

1 Introduction

This is reference documentation for the BHM and PPM *run time* function interface, defined in
`ImpPublic/include/target/peripheral/ppm.h`
and
`ImpPublic/include/target/peripheral/bhm.h`

The functions in this interface are used within code written and compiled for the Peripheral Simulation Environment (PSE).

2 Peripheral Interface Specification

A peripheral model must provide a structure describing its interface, which is part of the peripheral model executable (usually called `pse.pse`) and can be interrogated by the simulator before any peripheral code is executed. Recent versions of OVPsim and CpuManager require this structure to be present and complete.

The structure must be called `modelAttrs` and be of type `ppmModelAttr`.

2.1 Peripheral modelAttrs structure

Prototype

```
typedef struct ppmModelAttrS {

    ////////////////////////////////////////////////////////////////////////
    // VERSION and IDENTIFICATION
    ////////////////////////////////////////////////////////////////////////

    ppmString              versionString;    // Must be PPM_VERSION_STRING
    ppmModelType           type;             // MUST be set to PPM_PERIPHERAL

    ////////////////////////////////////////////////////////////////////////
    // Model status
    ////////////////////////////////////////////////////////////////////////

    ppmVisibility          visibility;           // model instance is invisible
    ppmReleaseStatus       releaseStatus;        // model release status (enum)
    Bool                   saveRestore;          // model supports save and restore
    Bool                   noRecursiveCallbacks; // a callback can be triggered
                                                 // from within this model

    ////////////////////////////////////////////////////////////////////////
    // Callbacks
    ////////////////////////////////////////////////////////////////////////

    ppmBusPortSpecFn       busPortsCB;       // next bus port callback
    ppmNetPortSpecFn       netPortsCB;       // next net port callback
    ppmPacketnetPortSpecFn packetnetPortsCB; // next net port callback
    ppmConnInputPortSpecFn connInputsCB;     // next FIFO input port
    ppmConnOutputPortSpecFn connOutputsCB;   // next FIFO input port
    ppmParameterSpecFn     paramSpecCB;      // next parameter callback
    ppmSaveStateFn         saveCB;           // PSE state save callback
    ppmRestoreStateFn      restoreCB;        // PSE state restore callback
    ppmDocFn               docCB;            // This function installs
                                             // documentation nodes

    ////////////////////////////////////////////////////////////////////////
    // Data needed by a simulator for peripheral model.
    ////////////////////////////////////////////////////////////////////////

    // Location of this model
    ppmVlnvInfo vlnv;

    // Optional Extension library used when the model requires native code
    ppmString   extension;

    // Path to PDF documentation
    ppmString    doc;

     // Model family string
    ppmString    family;

} ppmModelAttr, *ppmModelAttrP;
```

Description

Field `versionString` must be set to the macro `PPM_VERSION_STRING`. Field `type` must be set to the macro PPM_MT_PERIPHERAL.

Field `visibility` indicates whether details of the peripheral model should be exposed to a debugger. Values for this parameter are defined by type `ppmVisibility`, as follows:

```
typedef enum ppmVisibilityE {
    PPM_VISIBLE,
    PPM_INVISIBLE
```

```
} ppmVisibility;
```

Field `releaseStatus` is used for documentation only and indicates the release status of the model. Values for this parameter are defined by type ppmReleaseStatus, as follows:

```
typedef enum ppmReleaseStatusS {
    PPM_UNSET,
    PPM_INTERNAL,
    PPM_RESTRICTED,
    PPM_IMPERAS,
    PPM_OVP,
} ppmReleaseStatus;
```

Fields `busPortsCB`, `netPortsCB`, `packetnetPortsCB`, `connInputsCB`, `connOutputsCB` and `paramSpecsCB` are iterator function pointers described below.

Fields `saveCB` and `restoreCB` are used to define model-specific save and restore functions. These are used to checkpoint running simulations and restore such checkpoints at a later date.

Field `docCB` is used to add documentation to a peripheral model.

Field `vlnv` is a structure which describes where the model is stored in an Imperas VLNV tree.

Field `doc` describes the location of the model's documentation.

Field `family` is used by Imperas products.

Field `extension` is used if this peripheral uses a native code extension library. Normally the peripheral program and the extension library binaries are stored in the same directory. Set `extension` to the name of the extension library (without its file extension).

Field `noRecursiveCallbacks` controls what happens when the model reads or writes to a memory region in this model that has a callback associated with it.
If `True`, the callback will not occur, if `False`, it will. In most models, callbacks are expected to occur when *another* model writes to a sensitive region (e.g. a processor model writing to a register model in peripheral), so `noRecursiveCallbacks` should be `True`; however some models trigger their own callbacks on purpose, so need it to be `False`. Note that if this technique is used, the model must protect against infinite recursion – the simulator always detects this condition and stops the model after 15 levels of recursion.

2.2 Bus port definition

Prototype

```
#define PPM_BUS_PORT_FN(_name) ppmBusPortP   _name(ppmBusPortP busPort)
typedef PPM_BUS_PORT_FN((*ppmBusPortSpecFn));
```

Description

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

The `ppmBusPort` structure describes one bus port and contains these fields:

Type	Name	Description
`Addr`	`addrHi`	(slave port only) Size in bytes of the bus port, less one byte.
`const char *`	`name`	name of the port
`ppmBusPortType`	`type`	type of the port (see below)
`Uns32`	`addBits`	(master port only) Number of address bits implemented
`const char *`	`description`	For documentation
`Bool`	`mustBeConnected`	True if this port must be connected
`Bool`	`remappable`	(slave port only) True if the model moves the decode address at run-time.

Bus port types:

ppmBusPortType	Description
`PPM_MASTER_PORT`	Port which initiates transaction
`PPM_SLAVE_PORT`	Port which receives transactions

Example

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

2.3 Net port definitions

Prototype

```
#define PPM_NET_PORT_FN(_name) ppmNetPortP   _name(ppmNetPortP netPort)
typedef PPM_NET_PORT_FN((*ppmNetPortSpecFn));
```

Description

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

The `ppmNetPort` structure contains these fields:

Type	Name	Description
`const char *`	`name`	name of the port
`ppmNetPortType`	`type`	type of the port
`const char *`	`description`	For documentation
`Bool`	`mustBeConnected`	True if this port must be connected
`ppmNetFunc`	`netCB`	Pointer to function called when the net is written
`void *`	`userData`	Passed to the callback

Net port types:

ppmNetPortType	**description**
`PPM_INPUT_PORT`	Single wire input
`PPM_OUTPUT_PORT`	Single wire output

Example

```
// example

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

2.4 Packetnet port definitions

Prototype

```
#define PPM_PACKETNET_PORT_FN(_name) \
    ppmPacketnetPortP   _name(ppmPacketnetPortP packetnetPort)

typedef PPM_PACKETNET_PORT_FN((*ppmPacketnetPortSpecFn));
```

Description

A *packetnet* is an abstraction facilitating implementation of models of packet-based networks. See the *OVPsim and CpuManager User Guide* for more information about packetnets.

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

The `ppmPacketnetPort` structure 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
`ppmPacketnetFunc`	`packetnetCB`	Function called when packetnet is written
`void *`	`userData`	Passed to the callback
`uns32`	`sharedDataBytes`	Maximum number of bytes sent in one packet
`void *`	`sharedData`	Pointer to shared data area
`ppmPacketnetHandlePtr`	`handlePtr`	Pointer to handle, updated by simulator

Example

```
static PPM_PACKETNET_PORT_FN(nextPacketnetPort) {

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

2.5 Conn Input and output port definitions

Prototype

```
#define PPM_CONN_INPUT_FN(_name) \
    ppmConnInputPortP   _name(ppmConnInputPortP   port)

typedef PPM_CONN_INPUT_FN ((*ppmConnInputPortSpecFn));

#define PPM_CONN_OUTPUT_FN(_name) \
    ppmConnOutputPortP   _name(ppmConnOutputPortP   port)

typedef PPM_CONN_OUTPUT_FN ((*ppmConnOutputPortSpecFn));
```

Description

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

```
ppmConnInputHandle  port1Handle;
ppmConnOutputHandle port2Handle;

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

static ppmConnOutputPort connOutputPorts[] = {
```

```
    {
        .name            = “port1”,
        .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,
    // ...
};
```

2.6 Parameter definitions

Prototype

```
#define PPM_PARAMETER_FN(_name) ppmParameterP _name(ppmParameterP parameter)
typedef PPM_PARAMETER_FN((*ppmParameterSpecFn));
```

Description

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

Each returned structure describes one parameter. The `ppmParameter` structure contains these fields:

Type	Name	Description
`const char *`	`name`	parameter name
`ppmParameterType`	`type`	parameter type (see table)
`const char *`	`description`	short description
`type specifications`	`u`	union of possible type specifications
`void *`	`valuePtr`	pointer to a variable of the correct type

Each parameter type has a specification structure in a union which can be optionally set to check a parameter's value. If `valuePtr` is non-zero, it will be used as a destination for the value of the parameter with any assignment or override applied.

Parameter types:

`ppmParameterType`	Description	Type specification
`ppm_PT_BOOL`	boolean	default value
`ppm_PT_INT32`	signed 32b int	min, max and default value
`ppm_PT_UNS32`	unsigned 32b int	min, max and default value
`ppm_PT_INT64`	signed 64b int	min, max and default value
`ppm_PT_UNS64`	unsigned 64b int	min, max and default value
`ppm_PT_DOUBLE`	floating point number	min, max and default value
`ppm_PT_STRING`	const char *	optional max length and default value
`ppm_PT_ENUM`	enumerated type	array of legal values, 1st is default.
`ppm_PT_ENDIAN`	special enumerated type	default value

Example

```
// Example:

static PPM_PARAMETER_FN(nextParameter) {
    if(!parameter) {
        return parameters;
    }
    parameter++;
```

```
    return parameter->name ? parameter : 0;
}
```

2.7 Complete Example:

This is extracted from

`Examples/Models/Peripherals/creatingDMAC/4.interrupt/dmac.attrs.igen.c`

In this example, the bus and net structures are static. In a more complex model they could be generated dynamically.

```
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) {
        return busPorts;
    }
    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) {
        return netPorts;
    }
    netPort++;
    return netPort->name ? netPort : 0;
}

static ppmParameter parameters[] = {
    {
        .name        = "readNativeDataChannel",
        .type        = ppm_PT_BOOL,
        .description = "Use native code for DMA operation",
```

```
        .valuePtr    = &readNativeDataChannel,
    },
    { 0 }
};

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

ppmModelAttr modelAttrs = {

    .versionString = PPM_VERSION_STRING,
    .type          = PPM_MT_PERIPHERAL,

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

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

The model has one parameter - `readNativeDataChannel.` The boolean value, set by the platform or by `–override` on the simulator command line will be written into the variable `readNativeDataChannel` before `main()` is called in the peripheral model.

3 Behavioral Modeling *(BHM)*

This section describes functions which affect the execution of peripheral model code, and its interaction with the simulator.

3.1 bhmCreateThread

Prototype

```
typedef void  (*bhmCBThreadFunc)(void *user);

bhmThreadHandle bhmCreateThread(
    bhmCBThreadFunc cb,        // function which implements the thread
    void          *user,       // user data passed to the thread
    const char    *name,       // thread name (used for debugging)
    void          *sp          // optional address of the TOP of the stack
);
```

Description

This function creates a new thread. The return value is a handle to the thread which may be used to delete it.

A thread requires a stack. If this parameter is zero, a 1Mb stack is allocated. If more is required, a region must be reserved and the address of the TOP of the region passed to this parameter. The stack region must be 4-byte aligned.

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

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

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

Threads can be created or destroyed at any time.

Example

```
#include "peripheral/bhm.h”

bhmThreadHandle thA, thB;

typedef struct myThreadContextS {
    // store thread specific data here
} myThreadContext, myThreadContextP;

static BHM_THREAD_CB(myThread)
{
    while(1) {
        myThreadContextP context = userData;
        // can use context data here
        bhmWaitDelay(1000);
        bhmPrintf(“Running\n”);
    }
```

```
}

void userInit(void)
{
    thA = bhmCreateThread(myThread, malloc(sizeof(myThreadContext)), “threadA”, 0);
    thB = bhmCreateThread(myThread, malloc(sizeof(myThreadContext)), “threadB”, 0);
}
```

Notes and Restrictions

The stack should have sufficient space for that thread and any code it uses (libc can use a significant amount of stack).

The stack region must be 4-byte aligned.

3.2 *bhmThisThread*

Prototype

```
bhmThreadHandle bhmThisThread(void);
```

Description

This function returns the current thread. The return value is a handle to the thread which may be used to delete it.

Example

```
#include "peripheral/bhm.h”

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

static BHM_THREAD_CB(myThread)
{
    while(1) {
        bhmWaitDelay(1000);
        // print the name and handle from inside the thread
        bhmPrintf(“%s  h=%u\n”, (char*)userData, bhmThisThread());
    }
}

void userInit(void)
{
    // In this example a string is passed as the user data
    thA = bhmCreateThread(myThread, ”threadA”, “threadA”, 0);
    thB = bhmCreateThread(myThread, ”threadB”, “threadB”, 0);
}
```

Notes and Restrictions

bhmThisThread() must be called from within the thread.

3.3 bhmDeleteThread

Prototype

```
Bool bhmDeleteThread(bhmThreadHandle h);
```

Description

This function deletes an existing thread.

Example

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

// embedded call made on move to control status register
bhmThreadHandle  th = bhmCreateThread(myThread, NULL, “myThread”, &stack[size]);

bhmWait(1000*1000*1000);

bhmDeleteThread(th);
```

Notes and Restrictions

bhmDeleteThread can be called from within its own thread (which has the same effect as returning from the thread’s main function) or from another thread or callback. In the latter case the deleted thread must (by definition) have been blocked by a call to bhmWaitEvent() or bhmWaitDelay(), so the affect is as if the blocked call caused the thread to finish.

3.4 bhmCreateEvent

Prototype

```
bhmEventHandle bhmCreateEvent(void);
```

Description

This function creates an event object which can then be used by bhmWaitEvent(), bhmTriggerEvent(), bhmTriggerAfter() and bhmCancelTrigger().

Example

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

    bhmEventHandle  go_eh = bhmCreateEvent();
```

Notes and Restrictions

None.

3.5 bhmCreateNamedEvent

Prototype

```
bhmEventHandle bhmCreateNamedEvent(
    const char *name,
    const char *description
);
```

Description

This function creates an event object which can then be used by bhmWaitEvent(), bhmTriggerEvent(), bhmTriggerAfter() and bhmCancelTrigger().

A named event is similar to an un-named event, but is visible to the debugger, which can set trigger points on it. It should be used when the event might be meaningful to the user of the model.

Example

```
#include "peripheral/bhm.h”

bhmEventHandle go_eh = bhmCreateEvent("startDMA", "A DMA transfer has started");
```

Notes and Restrictions

None.

3.6 bhmDeleteEvent

Prototype

```
Bool bhmDeleteEvent(bhmEventHandle handle);
```

Description

This function deletes an event.

Example

```
#include "peripheral/bhm.h”

    bhmEventHandle  evt = bhmCreateEvent();

    bhmDeleteEvent(evt);
```

Notes and Restrictions

If an event is deleted when a thread is waiting for it, the thread will restart. The return code from bhmWaitEvent will be BHM_RR_DELEVENT.

3.7 bhmWaitEvent

Prototype

```
bhmRestartReason bhmWaitEvent(bhmEventHandle handle);
```

Description

The running thread stops until the event is triggered, the event is deleted or the event handle is invalid (this return is immediate).

Example

```
bhmEventHandle  ev1;

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

void thread2(void *user)
{
    while(1) {
   bhmWaitEvent(ev1);
    }
}
```

Notes and Restrictions

This function should not be called from a callback associated with a net, packetnet, diagnostic level or view object.

If called from a bus or register callback, a new thread is created. Please refer to OVP Peripheral Modelling Guide: “Delays in Callbacks”.

3.8 bhmTriggerAfter

Prototype

```
Bool bhmTriggerAfter(bhmEventHandle event, double delay);
```

Description

bhmTriggerAfter returns immediately but ‘queues’ a future trigger on the stated event. This queued trigger may be cancelled before the delay expires. If there is already a queued trigger, it will be replaced with the new one. Returns false if the handle was not valid.

Example

```
bhmEventHandle  ev1;

void thread1(void *user)
{
    while(1) {
        bhmWaitDelay(100 /*uS*/);
        bhmTriggerAfter(ev1, 20);
    }
}

void thread2(void *user)
{
    while(1) {
   bhmWaitEvent(ev1);
        // will run at times 120uS, 220uS, 320uS etc.
    }
}
```

Notes and Restrictions

None.

3.9 bhmCancelTrigger

Prototype

```
Bool bhmTriggerAfter(bhmEventHandle event);
```

Description

If an event is waiting to be triggered, bhmCancelTrigger cancels it and returns true. If not it returns false (a waiting event is created by calling bhmTriggerAfter). It returns false if the handle was not valid.

Example

```
bhmEventHandle  ev1;

void thread1(void *user)
{
    …
    bhmTriggerAfter(ev1, 20);
    …
}

void thread2(void *user)
{
    …
    bhmCancelTrigger (ev1);
    …
}
```

Notes and Restrictions

None.

3.10 bhmGetSystemEvent

Prototype

```
bhmEventHandle bhmGetSystemEvent(bhmSystemEventType eventType);
```

Description

Returns a handle to a system event. System event types include

```
BHM_SE_START_OF_SIMULATION
BHM_SE_END_OF_SIMULATION
```

Start of simulation occurs when all peripherals have executed their initialization code, but no application processors have executed any instructions.

End of simulation occurs when the simulator is performing a normal end of simulation sequence, i.e. there has not been a fatal error.

Example

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

int operationCount = 0;

main()
{
    bhmEventHAndle end = bhmGetSystemEvent(BHM_SE_END_OF_SIMULATION);


    ….
    bhmWaitEvent(end);
    bhmMessage("I", "MY_MODEL", "Finished after %d operations", operationCount);
}
```

Notes and Restrictions

BHM_SE_START_OF_SIMULATION need be used only when it is required that all other peripherals have started first.
Two peripherals waking on BHM_SE_START_OF_SIMULATION cannot rely on a particular order of execution.

3.11 bhmWaitDelay

Prototype

```
Bool bhmWaitDelay(double microseconds);
```

Description

Pauses the thread for the given time. Returns false if the request was unsuccessful.

The delay will be at least until the end of the current simulation time slice (aka quantum) because a time slice that has already started cannot be shortened.

If the delay time falls after the end of the current time slice then the time slice where the delayed time occurs will be adjusted so that the end of that time slice occurs at the exact time requested.

Excessive use of tiny delays in a peripheral model can thus have a similar effect on simulator performance as running with a very small time slice.

Example

```
#include "peripheral/bhm.h”


void thread1(void *user)
{
    while(1) {

        bhmWaitDelay(50);
        bhmMessage(“I”, “MY_MODEL”, “Starting…”);
        . . .
    }
}
```

Notes and Restrictions

1. This function should not be called from a callback associated with a net, packetnet, diagnostic level or view object.
2. If called from a bus or register callback, a new thread is created. Please refer to OVP Peripheral Modelling Guide: “Delays in Callbacks”.

3.12 bhmGetCurrentTime

Prototype

```
double bhmGetCurrentTime(void);
```

Description

Returns the current simulated time in microseconds.

The time returned is the simulation time at the beginning of the current simulation time slice (aka quantum). Thus multiple calls within the same time slice may observe time seeming to stand still, and peripherals cannot rely on resolving times shorter than the length of the simulation time slice.

Example

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

...
    bhmPrintf("The time is %0.0f\n", bhmGetCurrentTime());
...
```

Notes and Restrictions

Simulated time starts at zero each time a simulation begins and bears no relation to wallclock time.

3.13 bhmGetLocalTime

Prototype

```
double bhmGetLocalTime(void);
```

Description

Returns the current simulated time in microseconds from the perspective of the activating processor.

The time returned is the simulation time at the start of the current simulation time slice (aka quantum) plus a delta time based upon the activating processor instructions executed and MIPS rate. Thus multiple calls within the same time slice from a single processor will apparently show time moving forward. The time may not increase monotonically when a peripheral is accessed from multiple processors in the platform i.e. a call from another processor in the same time slice may show time apparently jumping backwards.

Example

```
#include "peripheral/bhm.h”

...
    bhmPrintf("The time is %0.0f\n", bhmGetLocalTime());
...
```

Notes and Restrictions

Simulated time starts at zero each time a simulation begins and bears no relation to wallclock time.

3.14 bhmMessage

Prototype

```
void bhmMessage(
    const char *severity,
    const char *prefix,
    const char *format,
    ...);
```

Description

Interface to the simulator text output and log streams. bhmMessage produces messages with the same format as simulator system messages. In addition, the *instance name* of the peripheral model is inserted into message so when multiple instances of the model are used, the programmer does not need to identify the particular instance.

Severity levels:
“I”	Information: nothing is wrong.
“W”	Warning: the simulation can continue normally.
“E”	Error: the simulation cannot proceed correctly.
“F”	Fatal: this will cause the simulator to exit after producing the message.

Prefix:
The prefix string has no format characters, so is guaranteed to appear verbatim in the output stream. It should be a short string (without spaces) making the message easy to distinguish from other output.

Format and varargs: conform to gnu libc printf.

Example

```
#include "peripheral/bhm.h”

{
    bhmMessage(“W”, “MY_PERIPH”, “Hello world number %d”, number);
}
```

Notes and Restrictions

bhmMessage will insert a new-line at the end of each message. To create tables and other formatted output, use `bhmPrintf()`.
Printing *long long ints* using the formats *%llx*, *%lld* and *%llu* in bhmMessage should be avoided due to a bug in the toolchain used to compile the PSE behavioral code.

3.15 bhmGetDiagnosticLevel

Superseded by bhmSetDiagnosticCB().

3.16 bhmSetDiagnosticCB

Prototype

```
void bhmSetDiagnosticCB(bhmSetDiagnosticLevel cb);
```

Description

Notifies the simulator that this function should be used to change the diagnostic level of the peripheral, indicating how much diagnostic output the model should produce. The simulator can set different levels for each instance of each model. Diagnostic output is intended to help *users* of the model (model developers can add *debug* output to their model, which should be hidden when the model is published). To ensure interoperability and easy familiarization, new models should conform to the following guidelines:

PSE Diagnostics (bits 0-1)

Level 0	No diagnostic output.
Level 1	Brief messages during startup (and possibly shutdown) to indicate the correct installation of the model in the platform.
Level 2	Comprehensive output; mode changes, complete operations, etc.
Level 3	Detailed output.

PSE Semihost Diagnostics (bits 2-3)

Level 0	No diagnostic output.
Level 1	Diagnostic output

System Diagnostics (bit 4)

Level 1	The simulator logs when it interacts with the model; e.g. when registers are read or written, when input nets change and when events are triggered.

To limit the size of log files, diagnostic levels can be changed during a simulation. Therefore the callback function should set the integer variable which is used to control diagnostic output

Example

```
#include "peripheral/bhm.h”

int diagLevel = 0;

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

int main()
{
    bhmSetDiagnosticCB(setDiags);

    if (diagLevel > 0) {
```

```
        bhmMessage(“I”, “MY_PERIPH”, “Starting up...”);
    }
}
```

Notes and Restrictions

Must be called before any diagnostic output is required. Should only be called once.

3.17 bhmPrintf

Prototype

```
void bhmPrintf(const char *format, ...);
```

Description

Send ‘raw’ characters to the text and log output streams. This function should only be used in conjunction with bhmMessage when tabular or formatted output is required. Unconstrained use will result in simulation messages whose ori*gi*n is hard to trace.

Example

```
#include "peripheral/bhm.h”

{
    bhmMessage(“I”, “MY_PERIPHERAL”, “Configuration:”);

    Uns32 I, J;
    for(I =0; I < height; I++) {
        bhmPrintf(“|”);
        for(J=0; J < width; J++) {
            bhmPrintf(“ %-4s”, config[I][J]);
        }
        bhmPrintf(“|\n”);
    }
}
```

Notes and Restrictions

Printing *long long ints* using the formats *%llx*, *%lld* and *%llu* formats in bhmPrintf should be avoided due to a bug in the toolchain used to compile the PSE behavioral code.

3.18 bhmFinish

Prototype

```
void bhmFinish(void);
```

Description

Terminate the simulation immediately. Normal shutdown procedures will be executed.

Example

```
#include "peripheral/bhm.h”

{
   if(noMoreData()) {
      if (BHM_DIAG_LOW) bhmMessage(“I”, “MY_MODEL”, “Data exhausted. Processing..”);
      bhmWaitDelay(50);
      if (BHM_DIAG_LOW) bhmMessage(“I”, “MY_MODEL”, “Finishing.”);
      bhmFinish();
    }
}
```

Notes and Restrictions

None.

3.19 Reading platform parameters

A model instance can change its behavior depending upon parameters set by the platform. For instance a single UART model could represent a 16450 or 16550 compatible device, the only difference being that the 16550 includes a FIFO in its data path. One of the two different behaviors could be selected by a boolean parameter. The parameter actual value is set on the instance in the platform.

Parameters are defined in the model's modelAttrs table – see section 2.6.

They are read using one of the following functions. Each function has the same semantics; the ptr parameter is written with the actual parameter's value from the platform, or if not specified in the platform, with the default value from the modelAttrs table. The function returns true if the parameter has been specified (or overridden) or false if the default value is being used.

Model parameters can be overridden from a control file or command line using the override command; refer to the OVP Control File User Guide.

Behaviour of Parameter Value functions:

condition	**action**	**returns**
value not set	ptr written with default value	false
value set in platform (no override)	ptr written with value from platform	true
value set by override	ptr written with value from override	true

3.19.1 bhmBoolParamValue

Prototype

```
Bool bhmBoolParamValue (const char *name, Bool *ptr);
```

Description

Read the value of a Boolean parameter.

Example

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

{
    Bool value;
    Bool isSpecified = bhmBoolParamValue ("myBooleanParameter", &value);
}
```

Note

The function can be used to read a parameter value but is not required. If the `valuePtr` field in the `ppmParameter` structure points to `value,` it will be written with the current parameter value.

3.19.2 bhmDoubleParamValue

Prototype

```
Bool bhmDoubleParamValue (const char *name, Bool *ptr);
```


Description

Read the value of a double (floating point) parameter.

Example

```
#include "peripheral/bhm.h”

{
    double value;
    Bool isSpecified = bhmDoubleParamValue ("myDoubleParameter", &value);
}
```

Note

The function can be used to read a parameter value but is not required. If the `valuePtr` field in the `ppmParameter` structure points to `value`, it will be written with the current parameter value.

3.19.3 bhmInt32ParamValue

Prototype

```
Bool bhmInt32ParamValue (const char *name, Bool *ptr);
```

Description

Read the value of an Int32 parameter.

Example

```
#include "peripheral/bhm.h”

{
    Int32 value;
    Bool isSpecified = bhmInt32ParamValue ("myInt32Parameter", &value);
}
```

Note

The function can be used to read a parameter value but is not required. If the `valuePtr` field in the `ppmParameter` structure points to `value`, it will be written with the current parameter value.

3.19.4 bhmInt64ParamValue

Prototype

```
Bool bhmInt64ParamValue (const char *name, Bool *ptr);
```

Description

Read the value of an Int64 parameter.

Example

```
#include "peripheral/bhm.h”

{
    Int64 value;
    Bool isSpecified = bhmInt64ParamValue ("myInt64Parameter", &value);
}
```

Note

The function can be used to read a parameter value but is not required. If the `valuePtr` field in the `ppmParameter` structure points to `value,` it will be written with the current parameter value.

3.19.5 bhmStringParamValue

Prototype

```
Bool bhmStringParamValue (const char *name, char *value, Uns32 maxLength);
```

Description

Read the value of an string parameter. The string value is copied into the given string. Any characters beyond the given maximum length are not copied.

Example

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

{
    char value[128];
    Bool isSpecified = bhmUns32ParamValue (
         "myStringParameter",
         value,
         sizeof(value)
    );
}
```

Note

The function can be used to read a parameter value but is not required. If the `valuePtr` field in the `ppmParameter` structure points to `value,` it will be written with the current parameter value.

3.19.6 bhmUns32ParamValue

Prototype

```
Bool bhmUns32ParamValue (const char *name, Bool *ptr);
```

Description

Read the value of an Uns32 parameter.

Example

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

{
    Uns32 value;
    Bool isSpecified = bhmUns32ParamValue ("myUns32Parameter", &value);
}
```

Note

The function can be used to read a parameter value but is not required. If the `valuePtr` field in the `ppmParameter` structure points to `value`, it will be written with the current parameter value.

3.19.7 bhmUns64ParamValue

Prototype

```
Bool bhmUns64ParamValue (const char *name, Bool *ptr);
```

Description

Read the value of an Uns64 parameter.

Example

```
#include "peripheral/bhm.h”

{
    Uns64 value;
    Bool isSpecified = bhmUns64ParamValue ("myUns64Parameter", &value);
}
```

Note

The function can be used to read a parameter value but is not required. If the `valuePtr` field in the `ppmParameter` structure points to `value`, it will be written with the current parameter value.

4 Record and Replay

If a peripheral model communicates with the outside world, e.g. through a real keyboard interface, a simulation might be affected by inputs which cannot be exactly reproduced in subsequent simulation sessions. This makes impossible regression testing or reproduction of particular failures. To overcome this problem, the bhm API presents a simple interface to a record/replay mechanism. It is the responsibility of the model writer to use this API if replay is required and to ensure that a model using replay does appear to the rest of the system to behave exactly as in the original simulation.

4.1 Overview

During startup (normally in 'main') the model should call `bhmRecordStart()` to see if recording is required by the simulator and if so, to start the recording. If recording is required, `bhmRecordEvent()` should be called whenever the model changes state due to external stimulus. Note that an event contains a time-stamp, a 'type' field which can be used to distinguish event types, and a variable length data field (which can be zero).

The model should also call `bhmReplayStart()` to see if this is a replay session. If so, the model should use `bhmReplayEvent()` to fetch each event, then act according to the event.

The location of the log data is managed by the simulation environment.

It is possible that (for testing), a model could both replay from a previous log and simultaneously record a new log.

Two record file formats are supported: a legacy binary format file (OVP1) and a new text-format file (OVP2). The simulator will read either format file, but by default writes the new format only. To force output in the legacy format, set the following environment variable:

```
IMPERAS_PSE_RECORD_VERSION=1
```

Imperas strongly recommend that the new format file should always be used.

4.2 Example

The PciIDE disk model in the intel.ovpworld.org directory supports record/replay. During initialization, function bdrv_open is called, which contains this code:

```
static Bool recording;
static Bool replaying;

BlockDriverStateP bdrv_open(Uns8 drive, const char *filename, Int32 flags)
{
    static Bool init = False;
    if (!init) {
        init = True;
        diag = bhmGetDiagnosticLevel();
        recording = bhmRecordStart();
        replaying = bhmReplayStart();
    }
    . . . lines deleted . . .
```

```
}
```

The initialization code sets static Booleans `recording` and `replaying` to indicate whether record mode and replay mode are active, respectively. Note that it is possible for both to be active simultaneously.

Each disk operation supported by the model is described in an enumeration:

```
typedef enum drEventTypeE {
    DR_OPEN = 1,        // open() call
    DR_CLOSE,           // close() call
    DR_READ,            // read() call
    DR_READ_DATA,       // read() data block
    DR_WRITE,           // write() call
    DR_FSTAT64,         // fstat64() call
    DR_FSTAT64_DATA,    // fstat64() data block
    DR_LSEEK64,         // lseek64() call
} drEventType;
```

There are functions which use the BHM primitives described in this document to record and replay an event of a particular type:

```
static void drRecordEventOfType(drEventType type, Uns32 bytes, void *data) {
    bhmRecordEvent(type, bytes, data);
}

static void drReplayEventOfType(drEventType type, Uns32 bytes, void *data) {

    drEventType actualType;
    Int32       actualBytes = bhmReplayEvent(NULL, &actualType, bytes, data);

    if(bytes<0) {
        bhmMessage("F", PREFIX,
            "Replay file ended: no further replay is possible"
        );
    } else if(type!=actualType) {
        bhmMessage("F", PREFIX,
            "Unexpected record type (required=%u, actual=%u)",
            type,
            actualType
        );
    } else if(bytes!=actualBytes) {
        bhmMessage("F", PREFIX,
            "Unexpected record size (required=%u, actual=%u)",
            bytes,
            actualBytes
        );
    }
}
```

Each supported primitive operation is wrapped by a utility routine that either implements the operation or replays it. If the operation is implemented, it is also recorded if required. For example, function `drRead` implements the basic `read` operation as follows:

```
static ssize_t drRead(Int32 fd, void *buf, size_t count) {

    ssize_t result;

    if(replaying) {
        drReplayEventOfType(DR_READ, sizeof(result), &result);
        if(result && (result!=-1)) {
            drReplayEventOfType(DR_READ_DATA, result, buf);
        }
```

```
    } else {
        result = read(fd, buf, count);
        if(recording) {
            drRecordEventOfType(DR_READ, sizeof(result), &result);
            if(result && (result!=-1)) {
                drRecordEventOfType(DR_READ_DATA, result, buf);
            }
        }
    }

    return result;
}
```

Function bdrvShutdown is called at the end of the simulation and includes code to close the record and replay files:

```
void bdrvShutdown(void)
{
    . . . lines deleted . . .

    bhmRecordFinish();
    bhmReplayFinish();
}
```

4.3 bhmRecordStart

Prototype

```
Bool bhmRecordStart(void);
```

Description

This function is called to determine if recording is required, and if so, prepare a recording channel for this model instance. It returns True if recording is required.

Example

This example is taken from the OVP PS2 Interface peripheral.

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

static Bool recording = False;

static Bool recordOpen(void)
{
    return (recording = bhmRecordStart());
}

void ps2Init(
    Bool     grabDisable,
    Bool     cursorEnable,
    updateFn keyboardCB,
    updateFn mouseCB
) {
    replayOpen();
    recordOpen();
    . . . etc . . .
}
```

Notes and Restrictions

1. The function must be called before any recordable events have occurred.
2. If the model also supports save/restore, record/replay state must be reestablished as part of the peripheral restore process. For the OVP PS2 Interface peripheral, this is done as follows:

```
void ps2Restore(void) {
    replayOpen();
    recordOpen();
}

PPM_SAVE_STATE_FN(peripheralSaveState) {
    // YOUR CODE HERE (peripheralSaveState)
}

PPM_RESTORE_STATE_FN(peripheralRestoreState) {
    ps2Restore();
}
```


4.4 bhmRecordEvent

Prototype

```
void bhmRecordEvent(Uns32 type, Uns32 bytes, void *data);
```

Description

This function records one event to the recording channel for this peripheral instance. The arguments are:

1. `type`: a model-specific event type code.
2. `bytes`: the size of the data associated with the event.
3. `data`: a pointer to the data block to be recorded.

If `bytes` is zero, this is a *null event* and the `data` argument is ignored.

Example

This example is taken from the OVP PS2 Interface peripheral.

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

typedef enum ktEventTypesE {
    KT_NULL = 78,
    KT_EVENT,
    KT_NO_MORE_EVENTS,
    KT_FINISH
} ktEventTypes;

static Bool recording = False;

static void recordNullEvent(void) {
    if(recording) {
        bhmRecordEvent(KT_NULL, 0, NULL);
    }
}

static void recordEvent(InputStateP is) {
    if(recording) {
        bhmRecordEvent(KT_EVENT, sizeof(*is), is);
    }
}

static void recordEndOfGroup(void)
{
    if(recording) {
        bhmRecordEvent(KT_NO_MORE_EVENTS, 0, NULL);
    }
}

static void livePoll(Bool disableInput) {

    if (disableInput) {
        return;
    }

    InputState  inputState;
    Uns32       iters = 0;

    while(kbControlPoll(&inputState,kbMouse)) {
        actOnEvent(&inputState);
        recordEvent(&inputState);
        iters++;
    }
```

```
    if(iters == 0)
        recordNullEvent();
    else
        recordEndOfGroup();
}
```

Notes and Restrictions

1. In a simulation in which both record and replay are active, it is not necessary to explicitly specify values to be recorded using `bhmRecordEvent`: the simulator automatically fills the record stream in this case, and calls to bhmRecordEvent are ignored.

4.5 bhmRecordFinish

Prototype

```
Bool bhmRecordFinish(void);
```

Description

Close the recording channel for this peripheral instance.

Example

This example is taken from the OVP PS2 Interface peripheral.

```
#include "peripheral/bhm.h”

void ps2Finish(void) {
    kbControlCleanUp();

    if(recording) {
        bhmRecordFinish();
    }
    if(replaying) {
        bhmReplayFinish();
    }
}
```

Notes and Restrictions

None.

4.6 bhmReplayStart

Prototype

```
Bool bhmReplayStart(void);
```

Description

This function is called to determine if replay is required, and if so, opens a channel for this model instance. The function returns True if replay is required.

Example

This example is taken from the OVP PS2 Interface peripheral.

```
#include "peripheral/bhm.h”

static Bool replaying = False;

static Bool replayOpen(void) {
    return (replaying = bhmReplayStart());
}

void ps2Init(
    Bool     grabDisable,
    Bool     cursorEnable,
    updateFn keyboardCB,
    updateFn mouseCB
) {
    replayOpen();
    recordOpen();
    . . . etc . . .
}
```

Notes and Restrictions

1. The function must be called before any replayable events have occurred.
2. If the model also supports save/restore, record/replay state must be reestablished as part of the peripheral restore process. For the OVP PS2 Interface peripheral, this is done as follows:

```
void ps2Restore(void) {
    replayOpen();
    recordOpen();
}

PPM_SAVE_STATE_FN(peripheralSaveState) {
    // YOUR CODE HERE (peripheralSaveState)
}

PPM_RESTORE_STATE_FN(peripheralRestoreState) {
    ps2Restore();
}
```

4.7 bhmReplayEvent

Prototype

```
Int32 bhmReplayEvent(double *time, Uns32 *type, Uns32 maxBytes, void *data);
```

Description

This function fetches the next event from the replay channel for this peripheral instance. It returns the number of bytes of user data associated with this event, which might be zero. A return value of -1 indicates that the end of the replay file has been reached and there are no more events to be read. Other arguments are as follows:

1. `time`: a by-ref argument filled with the time of this event. This parameter is for legacy use only and the returned value will always match the current simulated time when OVP2-format files are read. Pass `NULL` if the time is not required.
2. `type`: a by-ref argument filled with the model-specific event type code passed originally to bhmRecordEvent.
3. `maxBytes`: the maximum size of the data associated with the event. Simulation will exit with an error if the replayed data exceeds this size.
4. `data`: a pointer to a data block to be filled with data.

If the returned size is zero, this is a *null event* and the `data` argument is ignored.

Example

This example is taken from the OVP PS2 Interface peripheral.

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

static Bool replaying = False;

static void replayPoll(void) {

    static Bool fetch = True;

    while(fetch) {

        Uns32      type;
        InputState inputState;

        // get next event from replay file
        Int32 bytes = bhmReplayEvent(
            NULL, &type, sizeof(inputState), &inputState
        );

        if (bytes < 0) {

            // detect end-of-file
            fetch = False;

        } else {

            switch(type) {

                case KT_NULL:
                    return;

                case KT_EVENT:
                    actOnEvent(&inputState);
                    break;
```

```
                case KT_NO_MORE_EVENTS:
                    return;

                default:
                    bhmMessage("F", "PS2_IF", "Illegal entry in record file");
                    break;
            }
        }
    }
}

void ps2Poll(Bool disableInput) {

    if (replaying) {

        replayPoll();
        InputState inputState;
        kbControlPoll(&inputState, kbMouse);

    } else {

        livePoll(disableInput);
    }
}
```

Notes and Restrictions

1. It is the user's responsibility to ensure that the `data` buffer is large enough to handle any record type read from the replay file.
2. In a simulation in which both record and replay are active, it is not necessary to explicitly specify values to be recorded using `bhmRecordEvent`: the simulator automatically fills the record stream in this case, and calls to bhmRecordEvent are ignored.

4.8 bhmReplayFinish

Prototype

```
Bool bhmReplayFinish(void);
```

Description

Close the replay channel for this peripheral instance.

Example

This example is taken from the OVP PS2 Interface peripheral.

```
#include "peripheral/bhm.h”

void ps2Finish(void) {
    kbControlCleanUp();

    if(recording) {
        bhmRecordFinish();
    }
    if(replaying) {
        bhmReplayFinish();
    }
}
```

Notes and Restrictions

None.

4.9 Controlling record and replay

If you are using OVPsim, record or replay is turned on by defining the platform parameters `record` or `replay` on each peripheral instance that requires this behavior. It is usual to set record or replay for all peripherals or no peripherals so that the whole platform behaves consistently. See OVPsim and CpuManager User Guide for the definition of functions in this example:

```
#include "op/op.h"

optParamP      params = NULL;

if(replaymode) {
    params = opParamStringSet(params, "replay", getMyReplayFile());
}

optPeripheralP myPSE = opPeripheralNew (
    mi,
    psePath,
    “pse1”,
    OP_CONNECTIONS(
      OP_BUS_CONNECTIONS(
        OP_BUS_CONNECT(bus_b, "bp1", .slave=1, .addrLo=0x100, .addrHi=0x1ff)
      )
    ),
    params
);
```

If you are using the simulator with the standard command line parser, recording is turned on from the command line:

```
cmd> platform.<ARCH>.exe \
    .... \
    --modelrecorddir <directory>
    .... \
```

`<directory>` refers to a directory (folder) which will be created if it does not exist and in which the logged events will be stored. Explorer tags each file in the directory so it can check that the files are valid and that they match the platform.

Replay is similar; a directory (folder) is specified which contains pre-recorded events:

```
cmd> platform.<ARCH>.exe \
    .... \
    --modelreplaydir <directory>
    .... \
```

5 Platform Interaction (PPM)

PPM function provide access to the platform hardware; buses, bus-ports, nets and net-ports.

5.1 ppmOpenMasterBusPort

Prototype

```
ppmExternalBusHandle ppmOpenMasterBusPort(
    char          *busPortName,
    volatile void *localLoAddress,
    Uns64          sizeInBytes,
    SimAddr        remoteLoAddress
);
```

Description

Create a bus bridge from the PSE’s virtual address space to a simulated bus in the platform. Connection is by `busPortName` - the name of a master port in the peripheral model, which was connected to a bus during platform construction.

`localLoAddress` and `sizeInBytes` specify the connected region in the PSE’s address space.

`remoteLoAddress` specifies the address on the simulated bus that will be accessed from the first address in the connected region.

When a bus master port has been opened, reads and writes by the peripheral will be mapped to the simulated bus.

Returns a handle to the mapped region so it may be moved or unmapped later.

Example

```
#include “peripheral/ppm.h”


Uns8 masterRegion[1024];  // Local region to be mapped.

{
    ppmExternalBusHandle h = ppmOpenMasterBusPort(
        “portA”,
        &masterRegion[0],
        Sizeof(masterRegion),
        0x80000000
    );

    // This will fill with FFs the region 0x80000000 to 0x800003FF
    // on the bus connected to ‘portA’
    memset(masterRegion, 0xFF, sizeof(masterRegion));
}
```

Notes and Restrictions

1. The local region cannot be mapped more than once.
2. Reads and writes will be efficiently executed (as in the example, using memset) but cannot be accounted by bus traffic analysis tools or by simulation scheduling

algorithms which take account of bus traffic. To simulate discrete peripheral memory cycles, use ppmOpenAddressSpace.

5.2 ppmChangeRemoteLoAddress

Prototype

```
Bool ppmChangeRemoteLoAddress(
    ppmExternalBusHandle h,
    SimAddr remoteLoAddress
);
```

Description

Changes the remote address of an existing window.

Returns False if the operation fails.

Example

```
#include “peripheral/ppm.h”


Uns8 masterRegion[1024];  // Local region to be mapped.

{
    ppmExternalBusHandle h = ppmOpenMasterBusPort(
        “portA”,
        &masterRegion[0],
        Sizeof(masterRegion),
        0x80000000
    );

    // This will fill with FFs the region 0x80000000 to 0x800003FF
    // on the bus connected to ‘portA’
    memset(masterRegion, 0xFF, sizeof(masterRegion));

    ppmChangeRemoteLoAddress(h, 0x90000000);

    // This will fill with FFs the region 0x90000000 to 0x900003FF
    memset(masterRegion, 0xFF, sizeof(masterRegion));

}
```

Notes and Restrictions

None.

5.3 ppmOpenAddressSpace

Prototype

```
ppmAddressSpaceHandle ppmOpenAddressSpace(char *busPortName);
```

Description

Procedural access to simulated buses. Returns a handle to an address space which may be used to read and write directly to that space.

Returns `0` if the port does not exist.

Example

```
#include “peripheral/ppm.h”

{
    ppmAddressHandle h = ppmOpenAddressSpace(“portA”);
    if(!h) {
        // error handling
    }
    Uns8 buf[4];

    ppmReadAddressSpace(h, 0x80000000, sizeof(buf), buf);
    ppmWriteAddressSpace(h, 0x90000000, sizeof(buf), buf);

    ppmCloseAddressSpace(h);
}
```

Notes and Restrictions

None.

5.4 ppmReadAddressSpace

Prototype

```
Bool ppmReadAddressSpace(
    ppmAddressSpaceHandle  handle,
    Uns64                  address,
    Uns32                  bytes,
    void                   *data
);
```

Description

Atomic read of data from an address space into a local buffer.

Returns False if the operation fails.

Example

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

{
    ppmAddressHandle h = ppmOpenAddressSpace("portA");
    Uns8 buf[4];

    // copy 4 bytes from 0x80000000 - 0x80000003
    // to 0x90000000 - 0x90000003
    // on bus connected to portA
    ppmReadAddressSpace(h, 0x80000000, sizeof(buf), buf);
    ppmWriteAddressSpace(h, 0x90000000, sizeof(buf), buf);

    ppmCloseAddressSpace(h);
}
```

Notes and Restrictions

None.

5.5 ppmWriteAddressSpace

Prototype

```
Bool ppmWriteAddressSpace(
    ppmAddressSpaceHandle  handle,
    Uns64                  address,
    Uns32                  bytes,
    void                   *data
);
```

Description

Atomic write of data to an address space from a local buffer.

Returns False if the operation fails.

Example

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

{
    ppmAddressHandle h = ppmOpenAddressSpace("portA");
    Uns8 buf[4];

    // copy 4 bytes from 0x80000000 - 0x80000003
    // to 0x90000000 - 0x90000003
    // on bus connected to portA
    ppmReadAddressSpace(h, 0x80000000, sizeof(buf), buf);
    ppmWriteAddressSpace(h, 0x90000000, sizeof(buf), buf);

    ppmCloseAddressSpace(h);
}
```

Notes and Restrictions

None.

5.6 ppmTryReadAddressSpace

Prototype

```
Bool ppmTryReadAddressSpace(
    ppmAddressSpaceHandle  handle,
    Uns64                  address,
    Uns32                  bytes
);
```

Description

See if atomic read of data from an address space would complete or not.

Returns False if the operation would not complete.

Example

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

{
    ppmAddressHandle h = ppmOpenAddressSpace("portA");

    // try to read 4 bytes from 0x80000000 - 0x80000003
    // on bus connected to portA
    Bool ok = ppmTryReadAddressSpace(h, 0x80000000, 4);

    func(ok);

    ppmCloseAddressSpace(h);
}
```

Notes and Restrictions

None.

5.7 ppmTryWriteAddressSpace

Prototype

```
Bool ppmTryWriteAddressSpace(
    ppmAddressSpaceHandle  handle,
    Uns64                  address,
    Uns32                  bytes
);
```

Description

See if atomic write of data to an address space would complete or not.

Returns False if the operation would not complete.

Example

```
#include “peripheral/ppm.h”

{
    ppmAddressHandle h = ppmOpenAddressSpace(“portA”);

    // try write 4 bytes from 0x90000000 - 0x90000003

    Bool ok = ppmTryWriteAddressSpace(h, 0x90000000, 4);

    func(ok);

    ppmCloseAddressSpace(h);
}
```

Notes and Restrictions

None.

5.8 ppmCloseAddressSpace

Prototype

```
Bool ppmCloseAddressSpace(ppmAddressSpaceHandle h);
```

Description

Close an address space.

Returns False if the operation fails.

Example

```
#include “peripheral/ppm.h”

{
    ppmAddressHandle h = ppmOpenAddressSpace(“portA”);
    Uns8 buf[4];

    ppmReadAddressSpace(h, 0x80000000, sizeof(buf), buf);
    ppmWriteAddressSpace(h, 0x90000000, sizeof(buf), buf);

    ppmCloseAddressSpace(h);
}
```

Notes and Restrictions

None.

5.9 ppmOpenSlaveBusPort

Prototype

```
ppmLocalBusHandle ppmOpenSlaveBusPort(
    const char *portName,
    void       *localAddress,
    Uns64      sizeInBytes
);
```

Description

Expose a region in the PSE’s address space to reads and writes from a simulated bus. The local region effectively becomes RAM in the simulated system at the addresses specified in the construction of the port connection.

Returns 0 if the operation fails.

Example

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

{
    Uns8 rtcRam[32];

    ppmLocalBusHandle h = ppmOpenSlaveBusPort(“p1”, &rtcRam[0], sizeof(rtcRam));

    while(1) {
        bhmWaitDelay(1000 * 1000);

        if(++rtcRam[SECS] == 60) {
            rtcRam[SECS] = 0;
            if(++rtcRam[MINS] == 60) {
               rtcRam[MINS] = 0;
               if(++rtcRam[HRS] == 24) {
                  rtcRam[HRS] = 0;
               }
           }
        }
    }
}
```

Notes and Restrictions

The same area of memory can be exposed through more than one port, creating dual or multiple ported memories..

5.10 ppmCreateSlaveBusPort

Prototype

```
void *ppmCreateSlaveBusPort(
    const char *portName,
    Uns64      sizeInBytes
);
```

Description

Allocate a window of this many bytes and expose it to the bus connected to the named slave port. This function generally supersedes `ppmOpenSlaveBusPort()`, removing the need to allocate the window before exposing it. It is typically used in conjunction with `ppmCreateNByteRegister()` to create a set of memory-mapped registers which are accessible from a particular platform bus.

Example

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

{
    void *regPort = ppmCreateSlaveBusPort("regPort", 24);

    ppmCreateRegister("reg1", "control reg", regPort, 0,  4, .........);
    ppmCreateRegister("reg2", "data reg",    regPort, 4,  4, .........);

    .....
    ppmCreateRegister("reg6", "status reg",  regPort, 20, 4, .........);
}
```

Notes and Restrictions

This variant does not allow the moving (remapping) or deletion of the slave port. Use `ppmOpenSlaveBusPort()` is these facilities are required.

See `ppmCreateRegister`

5.11 ppmMoveLocalLoAddress

Prototype

```
Bool ppmMoveLocalLoAddress(
    ppmLocalBusHandle   h,
    void                *localAddress
);
```

Description

Move the exposed region in the PSE's address space.

Returns False if the operation fails.

Example

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

{
    Uns8 rtcRam[32];

    Uns8 backupRam[32];

    ppmLocalBusHandle h = ppmOpenSlaveBusPort(“p1”, &rtcRam[0], sizeof(rtcRam));


    bhmWaitDelay(1000 * 1000);

    if(backupMode()) {
        // now backupRam is exposed instead of rtcRam
        ppmMoveLocalLoAddress(h, &backupRam[0]);
    }
}
```

Notes and Restrictions

None.

5.12 ppmDeleteLocalBusHandle

Prototype

```
Bool ppmDeleteLocalBusHandle(ppmLocalBusHandle h);
```

Description

Delete a local mapped region.

Returns False if the operation fails.

Example

```
#include “peripheral/ppm.h”

{
    Uns8 rtcRam[32];

    ppmLocalBusHandle h = ppmOpenSlaveBusPort(“p1”, &rtcRam[0], sizeof(rtcRam));

    ...

    ppmDeleteLocalBusHandle(h);
}
```

Notes and Restrictions

None

5.13 ppmInstallReadCallback

Prototype

```
typedef Uns32(*ppmCBReadFunc)(void *addr, Uns32 bytes, void *user);

void ppmInstallReadCallback(
    ppmCBReadFunc cb,
    void *user,
    void *lo,
    Uns32 bytes
);
```

Description

(deprecated, use `ppmInstallNByteCallbacks()`)
Cause a user defined function to be called when a simulated processor or PSE reads from the specified region.

Arguments:

cb	the user function
user	user defined data which will be passed to the callback.
lo	base of the sensitized region
bytes	size of the sensitized region.

Example

```
#include “peripheral/ppm.h”

static Uns8 registers[4];

static PPM_READ_CB(readReg)
{
    If(bytes != 1) {
        bhmMessage(“F”, “MY_PERIPH”, “Only byte-wide access supported”);
    }
    Uns32 offset = (Uns8*)addr – registers;

    if(artifactAccess) {
        ...
    } else {
        switch(offset){
        case 0:
            return calcR0();
        case 1:
            return calcR1();
        case 2:
            return calcR2();
        default:
            return calcR3();
        }
    }
}

...{
    ppmOpenSlaveBusPort(“portA”, registers, sizeof(registers));
    ppmInstallReadCallback(readReg, NULL, registers, sizeof(registers));
    ppmInstallWriteCallback(writeReg, NULL, registers, sizeof(registers));
}
```

Notes and Restrictions

1. If the callback reads from it’s own sensitized region, a fatal recursion will occur.

2. A callback can be replaced by another on all or part of a region; the last install will win.
3. The callback should not call bhmWaitEvent() or bhmWaitDelay().
4. Use the prototype macro to declare the callback.
5. Delays in callbacks are allowed. Refer to OVP Peripheral Modelling Guide: “Delays in Callbacks”.
6. The callback supports a maximum transfer size of 4 bytes

5.14 ppmInstallWriteCallback

Prototype

```
typedef PPM_WRITE_CB((*ppmCBWriteFunc));

void ppmInstallWriteCallback(
    ppmCBWriteFunc cb,
    void *user,
    void *lo,
    Uns32 bytes
);
```

Description

(deprecated, use `ppmInstallNByteCallbacks`)
Cause a user defined function to be called when a simulated processor or PSE writes to the specified region.

Arguments:

cb	the user function
user	user defined data which will be passed to the callback.
lo	base of the sensitized region
bytes	size of the sensitized region.

Example

```
#include “peripheral/ppm.h”

static Uns8 registers[4];

static PPM_WRITE_CB(writeReg) {
    If(bytes != 1) {
        bhmMessage(“F”, “MY_PERIPH”, “Only byte-wide access supported”);
    }
    Uns32 offset = (Uns8*)addr – registers;
    if(artifactAccess) {
        // prevent side effects?
    } else {
        switch(offset){
        case 0:
            R0 = data;
             updateState();
             break;
        case 1:
            R1 = data;
             updateState();
             break;
        case 2:
            R2 = data;
             updateState();
             break;
        default:
            R3 = data;
             updateState();
             break;
        }
    }
}

...{
    ppmOpenSlaveBusPort(“portA”, registers, sizeof(registers));
    ppmInstallReadCallback(readReg, NULL, registers, sizeof(registers));
    ppmInstallWriteCallback(writeReg, NULL, registers, sizeof(registers));
```

```
}
```

Notes and Restrictions

1. If the callback writes to it’s own sensitized region, a fatal recursion will occur.
2. A callback can be replaced by another on all or part of a region; the last install will win.
3. The callback should not call bhmWaitEvent() or bhmWaitDelay().
4. Use the prototype macro to declare the callback.
5. The model writer might choose to inhibit side effects if the access is a simulation artifact.
6. Delays in callbacks are allowed. Refer to OVP Peripheral Modelling Guide: “Delays in Callbacks”.
7. The callback supports a maximum transfer size of 4 bytes

5.15 ppmInstallChangeCallback

Prototype

```
typedef PPM_WRITE_CB((*ppmCBWriteFunc));

void ppmInstallChangeCallback(
    ppmCBWriteFunc cb,
    void *user,
    void *lo,
    Uns32 bytes
);
```

Description

(deprecated, use `ppmInstallNByteCallbacks`)
Cause a user defined function to be called when a simulated processor or PSE writes a new value to the specified region.

Arguments:

cb	the user function
user	user defined data which will be passed to the callback.
lo	base of the sensitized region
bytes	size of the sensitized region.

Example

```
#include “peripheral/ppm.h”

static Uns8 registers[4];

static PPM_WRITE_CB(writeReg) {
    If(bytes != 1) {
        bhmMessage(“F”, “MY_PERIPH”, “Only byte-wide access supported”);
    }
    Uns32 offset = (Uns8*)addr – registers;
    if(artifactAccess) {
        // prevent side effects?
    } else {
        switch(offset){
        case 0:
            R0 = data;
            updateState();
            break;
        case 1:
            R1 = data;
            updateState();
            break;
        case 2:
            R2 = data;
            updateState();
            break;
        default:
            R3 = data;
            updateState();
            break;
        }
    }
}

{
    ppmOpenSlaveBusPort(“portA”, registers, sizeof(registers));
    ppmInstallChangeCallback(writeReg, NULL, registers, sizeof(registers));
}
```

Notes and Restrictions

1. If the callback writes to it’s own sensitized region, a fatal error will occur.
2. A callback can be replaced by another on all or part of a region; the last install will win.
3. The callback should not call bhmWaitEvent() or bhmWaitDelay().
4. Use the prototype macro to declare the callback.
5. The model writer might choose to inhibit side effects if the access is a simulation artifact.
6. Delays in callbacks are allowed. Refer to OVP Peripheral Modelling Guide: “Delays in Callbacks”.

5.16 ppmInstallNByteCallbacks

Prototype

```
/// Memory wide read callback
#define PPM_NBYTE_READ_CB(_name)  \
    void (_name)(                 \
        Uns32       offset,       \
        void       *data,         \
        Uns32       bytes,        \
        void       *userData,     \
        Bool        artifactAccess)


typedef PPM_NBYTE_READ_CB((*ppmNByteReadFunc));

/// Memory write write callback
#define PPM_NBYTE_WRITE_CB(_name) \
    void (_name)(                 \
        Uns32       offset,       \
        const void *data,         \
        Uns32       bytes,        \
        void       *userData,     \
        Bool        artifactAccess)

typedef PPM_NBYTE_WRITE_CB((*ppmNByteWriteFunc));

void ppmInstallNByteCallbacks(
    ppmNByteReadFunc  read,
    ppmNByteWriteFunc write,
    void             *userData,
    void             *loAddr,
    Uns32             bytes,
    Bool              readable,
    Bool              writable,
    Bool              isVolatile,
    bhmEndian         endian
);
```

Description

Cause user defined functions to be called when a simulated processor or PSE reads or writes to the specified region. Usually used with `ppmCreateSlaveBusPort()`. In the example below the slave port `sp1` will be connected to a bus at an address set by code in the platform (or code in module which is part of the platform). The region returned by `ppmCreateSlaveBusPort()` will be mapped to the region on the connected bus.

Arguments:

read	The user function called when the region is read
write	The user function called when the region is written
userData	User defined data which will be passed to the callback.
loAddr	Base of the sensitized region
bytes	Size of the sensitized region.
readable	If true, the region can be read. Must be true if read function is specified. If false and there is no callback then the region is protected against reading.
writeable	If true, the region can be written. Must be true if write function is specified. If false and there is no callback then the region is protected against writing.

isVolatile	If true, region will be written if the value has changed. If false, writes of the same value are suppressed.
endian	Controls byte-swapping. If `BHM_ENDIAN_BIG,` values will be byte-swapped.

Example

```
#include “peripheral/ppm.h”

PPM_NBYTE_READ_CB(readNCB) {
    const char *txt = userData;
    bhmPrintf(
        "Read %s bytes=%u user=%s  offset=%u\n",
        artifactAccess ? “artifact” : “real”,
        bytes,
        txt,
        offset
    );
    Uns32 b;
    Uns8 *p;
    for(p = data, b = 0; b < bytes; b++) {
        *p++ = nextByte();
    }
}

PPM_NBYTE_WRITE_CB(writeNCB) {
    const char *txt = userData;
    bhmPrintf(
        "Write %s bytes=%u user=%s  offset=%u\n",
        artifactAccess ? “artifact” : “real”,
        bytes,
        txt,
        offset
    );

    Uns32 b;
    const Uns8 *p;
    for(p = data, b = 0; b < bytes; b++) {
        bhmPrintf(" 0x%x", *p++);
    }
    bhmPrintf("\n");
}

int main() {
    Uns32 bytes = 64;
    void *handle = ppmCreateSlaveBusPort("sp1", bytes);
    ppmInstallNByteCallbacks(
        readNCB,
        writeNCB,
        "read write",
        handle,
        bytes,
        1,
        1,
        0,
        endian
    );
}
```

Notes and Restrictions

1. If a callback writes to it’s own sensitized region, a fatal error will occur.
2. A callback can be replaced by another on all or part of a region; the last install will win.
3. The callback should not call `bhmWaitEvent()` or `bhmWaitDelay()`.
4. Use the prototype macro to declare the callback.

5. The model writer might choose to inhibit side effects if the access is a simulation artifact.
6. If the `read` callback is specified, the `readable` flag is ignored and assumed to be true.
7. If the `write` callback is specified, the `writable` flag is ignored and assumed to be true.
8. Delays in callbacks are allowed. Refer to OVP Peripheral Modelling Guide: “Delays in Callbacks”.
9. To signal that the read or write cannot be completed, the callback should call `ppmReadAbort` or `ppmWriteAbort`.

5.17 ppmReadAbort

Prototype

```
void ppmReadAbort(void);
```

Description

In a bus or register read callback, signal that the read cannot be completed. This will abort the read in the application processor that initiated it.

5.18 ppmWriteAbort

Prototype

```
void ppmWriteAbort(void);
```

Description

In a bus or register write callback, signal that the write cannot be completed. This will abort the read in the application processor that initiated it.

5.19 ppmOpenNetPort

Prototype

```
ppmNetHandle ppmOpenNetPort(const char *portName);
```

Description

Makes a connection to the net connected to the given net port.

Returns a handle to the net.

A net is a means of connecting a function call in one model (the net driver) to a function call-back in one or more other models (the receivers). The net does not model contention; a net takes the last written value. Writing a net with the same value **will** cause call-backs to occur. A net value is a 32-bit integer; it can mean whatever the user wishes, but typically takes the values zero or non-zero to mean logic 0 or 1.

Example

```
#include “peripheral/ppm.h”

ppmNetHandle h = ppmOpenNetPort("int2");

void raiseInt(void) {
    ppmWriteNet(h, 1);
}

void lowerInt(void) {
    ppmWriteNet(h, 0);
}
```


5.20 ppmWriteNet

Prototype

```
void ppmWriteNet(ppmNetHandle handle, ppmNetValue value);
```

Description

Propagate a value to all ports connected to the given net. Any other connected port will cause its net callbacks to occur.

Example

```
#include “ppm.h”

ppmNetHandle h = ppmOpenNetPort("int2");

void raiseInt(void) {
    ppmWriteNet(h, 1);
}

void lowerInt(void) {
    ppmWriteNet(h, 0);
}
```

Notes and Restrictions

1. There is no simulation of net contention; the net takes the last value written.
2. Care should be taken if ppmWriteNet() is called from a net callback; A cyclic net topology in the platform will cause a fatal recursion.

5.21 ppmReadNet

Prototype

```
ppmNetValue ppmReadNet(ppmNetHandle handle);
```

Description

Returns the last value written to the net.

Example

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

{
    ppmNetHandle h = ppmOpenNetPort("int2");
}

ppmNetValue old = 0;

void checkNet(void) {
    ppmNetValue new = ppmReadNet(h);
    if (new != old) {
        bhmMessage("I", "NY_PERIPHERAL", "Net 'int2' changed");
        old = new;
    }
}
```

Notes and Restrictions

None.

5.22 ppmInstallNetCallback

Prototype

```
typedef PPM_NET_CB((*ppmCBNetFunc));

void ppmInstallNetCallback(
    ppmNetHandle  handle,
    ppmCBNetFunc  cb,
    void          *userData
);
```

Description

Install a function callback on a net. The function will be called when any device writes a value to the net (even if the new value is same as the current value).

Example

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

ppmNetValue old = 0;

PPM_NET_CB(netChanged) {
    if (new != old) {
        bhmMessage("I", "NY_PERIPHERAL", "Net 'int2' changed");
        old = new;
    }
}

...{
    ppmNetHandle h = ppmOpenNetPort("int2");
    ppmInstallNetCallback(h, netChanged, 0);
}
```

Notes and Restrictions

1. The callback should not call bhmWaitEvent() or bhmWaitDelay().
2. Use the prototype macro PPM_NET_CB to declare the callback.

5.23 ppmCreateDynamicBridge

Prototype

```
Bool ppmCreateDynamicBridge(
    const char  *slavePort,
    SimAddr     slavePortLoAddress,
    Uns64       windowSizeInBytes,
    const char  *masterPort,
    SimAddr     masterPortLoAddress
);
```

Description

Create a region of windowSizeInBytes starting at slavePortLoAddress 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 masterPortLoAddress.

Example

```
#include "peripheral/ppm.h"
...{
    ppmCreateDynamicBridge( "sp1", 0x40000000, 0x1000, "mp1", 0x100);
}
```

Notes and Restrictions

1. The region formed by windowSizeInBytes starting at slavePortLoAddress must not overlap any other static or dynamic decoded region on the connected bus.
2. The region formed by windowSizeInBytes starting at masterPortLoAddress can overlap other master regions, resulting in shared or 'dual ported' regions.

5.24 ppmDeleteDynamicBridge

Prototype

```
void ppmDeleteDynamicBridge(
    const char *slavePort,
    SimAddr    slavePortLoAddress,
    Uns64      windowSizeInBytes
);
```

Description

Delete a previously constructed Dynamic Bridge of windowSizeInBytes starting at slavePortLoAddress on the bus connected to slavePort. Reads or writes by bus masters on this bus to this region will be no longer mapped to another bus.

Example

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

...{
    ppmCreateDynamicBridge( "sp1", 0x40000000, 0x1000, "mp1", 0x100);
}

...{
    ppmDeleteDynamicBridge( "sp1", 0x40000000, 0x1000);
}
```

Notes and Restrictions

1. Only use this to remove a complete region created by ppmCreateDynamicBridge.
2. Do not attempt to split a region by un-mapping part of an existing region.
3. Do not attempt to un-map a region created by other means.

5.25 ppmCreateDynamicSlavePort

Prototype

```
void ppmCreateDynamicSlavePort(
    const char *slavePort,
    SimAddr    slaveLoAddress,
    Uns64      sizeInBytes,
    void       *localLowAddress
);
```

Description

Expose the local region `localLowAddress` of size `sizeInBytes` starting at `slavePortLoAddress` to the remote bus connected to `slavePort`. Reads or writes by bus masters on the remote bus will be mapped to the local region.

Example

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

unsigned char   remappedRegion[sizeInBytes]; // allocate an area to be read/written

const    char   *portName = "sp1";
static   SimAddr loAddr   = initialAddress();  // remember port addr

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

Notes and Restrictions

1. Do not overlap the remote region with any other static or dynamically mapped devices.
2. More than one mapping can be made onto the local region, to give dual port behavior or to model folding caused by (for example) incomplete address decoding.

5.26 ppmDeleteDynamicSlavePort

Prototype

```
void ppmCreateDynamicSlavePort(
    const char *slavePort,
    SimAddr    slavePortLoAddress,
    Uns64      sizeInBytes
);
```

Description

Remove a mapping that was created using `ppmCreateDynamicSlavePort`.

Example

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

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

ppmCreateDynamicSlaveBusPort(  // set the initial port address
    portName,
    loAddr,
    remappedRegion,
    sizeInBytes
);
ppmDeleteDynamicSlavePort(       // remove the old mapping
    portName,
    loAddr,
    sizeInBytes
);
```

Notes and Restrictions

1. Do not use this function to remove any other kind of mapped region.

6 Memory mapped registers

6.1 ppmCreateRegister

Prototype

```
registerHandle ppmCreateRegister(
    const char     *name,          // name of register
    const char     *description,   // to appear in the debugger
    void           *base,          // base of local window
                                   // (returned by ppmCreateSlaveBusPort)
    Uns32          offset,         // from base of window
    Uns64          bytes,          // size of this register
    ppmCBReadFunc  readCB,         // called by a bus read
    ppmCBWriteFunc writeCB,        // called by a bus write
    ppmCBviewFunc  viewCB,         // called by debugger to non-destructively
                                   // fetch the current value
    void          *userData        // will be passed to the 3 callbacks.
    Bool           isVolatile      // if false, writes of the same value will be
                                   // optimized away
);
```

Description

(deprecated, use `ppmCreateNByteRegister`)

Similar to `ppmInstallReadCallback` and `ppmInstallWriteCallback`, but additionally creates an object visible to the debugger. The register-object has a name and description. It is accessed by a bus access of the correct size.

The debugger can view the register without changing its value (which might occur if the register is read by a regular bus access, e.g. at the debug prompt: print /x *myRegisterPointer) using the viewCB function.

The register has debugger trigger-events associated with bus reads and writes.

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.

Example

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

static PPM_READ_CB(readCB) {
    if (!artifactAccess) {
        readDone();   // side-effect of the read
    }
    return reg1;
}

static PPM_WRITE_CB(writeCB) {
    reg1 |= data;   // write behavior need not be straight-forward
}

static PPM_VIEW_CB(viewCB) {
    *(Uns32*)data = reg1;    // return the true value without side effects
}
```

```
void installRegs (){
    void *regPort = ppmCreateSlaveBusPort("regPort", 24);

    ppmCreateRegister(
        "reg1",                  // name
        "control register1",     // description
         regPort,                // base of window
         0,                      // offset from window base
         4,                      // size in bytes
         readCB,                 // bus read function
         writeCB,                // bus write function
         viewCB,                 // debugger view function
         True                    // volatile register
    );

    ppmCreateRegister(
        "reg2",                  // name
        "control register2",     // description
         regPort,                // base of window
         4,                      // offset from window base
         4,                      // size in bytes
         readCB,                 // bus read function
         writeCB,                // bus write function
         viewCB,                 // debugger view function
         False                   // non-volatile register
    );
}
```

In the example, 'reg1' occupies the first 4 bytes of the 24-byte port. The register callback will occur whenever a write occurs to its location, regardless of value; 'reg2' occupies the next 4 bytes. The register callback will not occur when the same value is re-written.

The remaining 16 bytes of the window are implemented by memory which was allocated by the call to `ppmCreateSlaveBusPort()`.

Notes and Restrictions

1. Registers should not overlap.
2. Use the prototype macro to declare the callback.
3. Register size is limited to a maximum of 4 bytes.

6.2 ppmCreateNByteRegister

Prototype

```
registerHandle ppmCreateNByteRegister(
    const char     *name,           // name of register
    const char     *description,    // to appear in the debugger
    void           *base,           // base of local window
                                    // (returned by ppmCreateSlaveBusPort)
    Uns32          offset,          // from base of window
    Uns64          bytes,           // size of this register
    ppmNByteReadFunc  readCB,     // called by a bus read
    ppmNByteWriteFunc writeCB,    // called by a bus write
    ppmNByteViewFunc  viewCB,     // called by debugger to non-destructively
                                    // fetch the current value
    void          *data,            // where the data is stored
    void          *userData,        // will be passed to the 3 callbacks.
    Bool           isVolatile,      // if false, writes of the same value will be
                                    // optimized away
    Bool           readable,        // true if register can be read
    Bool           writable,        // true if register can be written

    bhmEndian      endian           // if big endian, data will be byte-swapped
);
```

Description

Creates a memory mapped register object which is also visible to the debugger.

The recommended way to model memory mapped registers is to reserve a region of memory in the peripheral model's address space (the *window*) which will be mapped to a region on a simulated bus. Registers are then installed with different offsets from the base of the window. 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.

Setting readable False and not supplying `readCB` will cause the accessing device to simulate a bus error if a read from the register attempted. Setting writable False not supplying `writeCB` will cause a bus error if a write to the register is attempted.

Without a `readCB` callback, data will be read by the simulator from the storage referenced by the `data` pointer. A read larger than the `bytes` parameter is illegal. A read of fewer bytes will access part of the data.

Without a `writeCB` callback, data will be written to the storage referenced by the `data` pointer. A write larger than the `bytes` parameter is illegal. A write of fewer bytes will modify part of the data.

Without a `viewCB` callback, by a debugger from the storage referenced by the `data` pointer. A read of fewer bytes will access part of the data.

Callbacks

If a read or write requires side effects then the `readCB` and or `writeCB` must be supplied.

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 `data`, then `viewCB` must also be supplied. The `viewCB` is used by the debugger so must not cause side-effects.

Masking

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

Diagnostics

The register has debugger trigger-events associated with bus reads and writes. 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.

Endian

If the `endian` parameter is not the same as the host, then the simulator will byte-swap data supplied to and from the callbacks (if supplied) or byte-swap data as it is read or written to the `data` pointer.
Swapping is always byte-wise (b0,b1,b2,b3 becomes b3,b2,b1,b0).

Example

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

Uns32 reg1, reg2;

void installRegs (){
    void *regPort = ppmCreateSlaveBusPort("regPort", 24);

    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
    );

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

    // access to the part of the port that has no registers
```

```
    Uns8 *otherMemory = regPort;
    otherMemory[8] = ...
    ... = otherMemory[23];
```

In the example, `reg1` occupies the first 4 bytes of the 24-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 may or may not use the `reg2` variable.
The remaining 16 bytes of the window have no visible registers but will appear on the simulated bus. This can be accessed by code in the peripheral via (for example) the `otherMemory` pointer.

Notes and Restrictions

1. Registers should not overlap.
2. To declare the callbacks, use the prototype macros
   `PPM_NBYTE_READ_CB`
   `PPM_NBYTE_WRITE_CB`
   `PPM_NBYTE_VIEW_CB`
3. Callback functions must validate the size (number of bytes) of each access and act accordingly.
4. If the model is programmable endian, see `bhmEndianParamValue()`
5. A non-volatile register will not be written if the new value is known to be the same as the old – this allows the code generator to make optimizations.
6. Delays in callbacks are allowed. Refer to OVP Peripheral Modelling Guide: "Delays in Callbacks".
7. To signal that the read or write cannot be completed, the `readCB` or `writeCB` functions should call `ppmReadAbort` or `ppmWriteAbort`.

6.3 ppmCreateRegisterField

Prototype

```
void ppmCreateRegisterField(
    registerHandle  reg,            // handle from the containing register
    const char     *name,           // name of register field
    const char     *description,    // to appear in documentation
    Uns32          offset,          // from the LSB of the register
    Uns32          bits,            // size of this field
    Bool           read,            // can be read
    Bool           write            // can be written
);
```

Description

Create the description of a register bit field. A field describes a particular range of bits in a peripheral memory mapped register. Fields are (or will be) visible to the debugger. Fields allow read and write masking to be implemented without callbacks..

Example

```
#include "peripheral/ppm.h”
```

```
Uns64 reg1;

reg = ppmCreateNByteRegister(
        "reg1",                 // name
        "control register",     // 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

         False                  // not a volatile register
    );

//
// Define bits 4 to 5
//
ppmCreateRegisterField(
        reg,                    // containing register
        "field",                // name
        "The first field",      // description
         4,                     // offset from LSB
         2,                     // number of bits
         True                   // can be read
         True,                  // can be written
);
```

Masking

For registers created using `ppmCreateNByteRegister()`, each bit-field access permission creates automatic masking of the `data` variable:

- if `readCB` is not supplied, when the register is read, bitfields with the `read` parameter false will be read as zero.
- if `writeCB` is not supplied, when the register is written, bitfields with the `write` parameter false will be unchanged.

Notes and Restrictions

1. Register field numbers range from zero to the width of the register in bits, less one.
2. Register field number zero is the LSB.
3. Register fields must not overlap.
4. Register fields must not lie beyond width of the register
5. Register fields can be created in any order.

6.4 ppmCreateInternalRegister

Prototype

```
void *ppmCreateInternalRegister(
    const char     *name,           // name of register
    const char     *description,    // to appear in the debugger
    Uns64          bytes,           // size of this register
    ppmCBviewFunc  viewCB,          // called by debugger to non-destructively
                                    // fetch the current value
    void           *userData        // will be passed to the callback.
);
```

Description

(deprecated, use `ppmCreateNByteInternalRegister`)

Similar to `ppmCreateRegister;` creates a register with no direct bus access. Can be used, for example, to implement a register which is accessed via an index counter.

The debugger can view the register but cannot set a trigger point on its changing (since no read or write occurs).

Example

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

static PPM_VIEW_CB(viewCB) {
    *(Uns32*)data = reg1;    // return the value without side effects
}


void installRegs (){
    ppmCreateInternalRegister(
        "reg1",                  // name
        "control register",      // description
         4,                      // size in bytes
         viewCB                  // debugger view function
         0
    );
}
```

6.5 ppmCreateInternalNByteRegister

Prototype

```
void *ppmCreateInternalNByteRegister(
    const char      *name,            // name of register
    const char      *description,     // to appear in the debugger
    Uns64           bytes,            // size of this register
    ppmCBviewFunc   viewCB,           // called by debugger to non-destructively
                                      // fetch the current value
    void           *data,
    void           *userData          // will be passed to the callback.
);
```

Description

Similar to `ppmCreateRegister;` creates a register visible to the debugger but with no direct bus access.

The debugger can view the register but cannot set a trigger point on its changing (since no read or write occurs).

Example

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

Uns32 regInt;

void installRegs (){
    ppmCreateInternalNByteRegister(
        "regInt",                   // name
        "internal register",        // description
         sizeof(regInt),            // size in bytes
         0,                         // debugger view function
        &regInt,                    // storage
         0                          // not used
    );
}
```

7 Direct Bus Access

This interface allows direct access to a simulated bus without use of a port. Since the model needs to know the name of the bus to connect to, it's use is not recommended for re-usable models.

7.1 ppmAccessExternalBus

Prototype

```
ppmExternalBusHandle ppmAccessExternalBus(
    char              *remoteBusName,
    volatile void     *localLoAddress,
    Uns64             sizeInBytes,
    SimAddr           remoteLoAddress
);
```

Description

Create a bridge from a local (PSE) memory region to the simulated bus with the given name. Reads and writes by the PSE to the local region will be mapped to the region of the external bus. Note that ppmChangeRemoteLoAddress can be used to subsequently move the remote region.

Arguments:

remoteBusName	name of the simulated bus.
localLoAddress	PSE address of the base of the region
sizeInBytes	of the region
remoteLoAddress	base of the region on the simulated bus.

Example

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

Uns8 readWriteRegion[1024];

...{
    ppmExternalBusHandle h = ppmAccessExternalBus(
        "systembus",
        readWriteRegion,
        sizeof(readWriteRegion),
        0x80000000
    );
    memset(readWriteRegion, 0, sizeof(readWriteRegion));
}
```

Notes and Restrictions

Regions should not overlap on the local or remote buses.

7.2 ppmExposeLocalBus

Prototype

```
ppmLocalBusHandle ppmExposeLocalBus (
    char        *remoteName,
    SimAddr     remoteLoAddress,
    Uns64       sizeInBytes,
    void        *localLoAddress
);
```

Description

Create a bridge that exposes a region of the local (PSE) address space to a simulated address space. The name of the bus and address region on the bus is specified in this call rather than by a port connection, i.e. there is no port declared in the peripheral.

Example

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

Uns8 graphicsRam[4094];

...{
    ppmExternalBusHandle h = ppmExposeLocalBus(
        "systembus",
        graphicsRam,
        sizeof(graphicsRam),
        0x80000000              // graphics ram mapped here
    );
    while(1) {
        bhmWaitDelay(frameUpdatePeriod);
        updateDisplay(graphicsRam, sizeof(graphicsRam));
    }
}
```

Notes and Restrictions

A single region of PSE memory *can* be exposed more than once
The regions to where it is exposed cannot overlap.

8 Packetnet Interface

Models that communicate with Ethernet, USB CAN, GSM etc. can use the packetnet abstraction of a packet based network. A packet transaction is modeled as an instantaneous event; network speed and latency must be modeled in the transmitting or receiving devices. A packetnet communicates by callbacks and shared memory. The transmitting model creates a packet in its local memory then calls the transmit function. This causes a notification function to be called in each receiving model in turn, passing a pointer to and number of bytes in the packet. The notification function can modify the data if required. When every notification function has returned, the transmit function returns, then the transmitting model can examine the packet if required.

Note that peripheral models each occupy their own address spaces. Therefore the simulator copies the data as and when required, so the models must not rely on pointers in the data. The contents of a received packet should not be used after the notification function has returned.

The order that the connected models receive a packet is determined by the order of construction in the C code, but should not be relied on.

The peripheral model API can send and receive through the packetnet interface.
Packetnet Direction
A packetnet is bidirectional; a model can send and receive from the same packetnet (though it does not have to).

8.1 Packetnet ports

A named packetnet port represents the connection between a packetnet and a peripheral model instance.

8.2 Recursion

Common to several methods of communication between models, it is possible by carelessly connecting packetnets to create a loop so that a call in one model results in a call back into the same function in that model. The simulator detects and prevents deep recursion on any packetnet.

A peripheral model will not receive notification for a packet that it is sending.

8.3 Packet size

Physical networks have a maximum packet size. Larger data are broken into smaller units handled by the protocol stack. A peripheral model must specify the maximum number of bytes to be sent in one packet when it connects to a packetnet, though it can send fewer bytes if needed. All peripheral models on one packetnet must define the same maximum size. It is an error for the test-bench to transmit a packet larger than the size set by peripherals on the packetnet.

8.4 Packetnet functions

Send a packet to all receivers on a packetnet:

```
void ppmPacketnetWrite(ppmPacketnetHandle h, void *data, Uns32 bytes)
```

Note that a pointer to the handle appears in the packetnet port definition structure that is returned to the simulator by the packetnet port iterator function.

This defines the packetnet notification callback used to notify this peripheral model when a packet has arrived.

```
static PPM_PACKETNET_CB(receivePacketnet) {
   ...
}

/* receivePacketnet goes in the ppmPacketnetFunc field of the packetnetport
structure */
```

The function pointer goes in the ppmPacketnetFunc field of the packetnetport definition structure.

8.5 Example

An example using a packetnet is in:

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

9 Conn (FIFO) Support

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 2.5.

A peripheral model puts data into a FIFO or gets data out using a polled interface. To prevent unnecessary polling it can bind a FIFO to an event and wait on the event which is triggered when the FIFO has space or has data available. A peripheral can query a FIFO to determine its dimensions, connections and how much data is currently in the FIFO. The following functions provide this capability.

Function	**Port**	**Description**
`ppmConnPut`	output	Put a word into a conn if space is available
`ppmConnGet`	input	Get a word from a conn id data is available
`ppmRegisterConnOutputEvent`	output	bind an output port to an event
`ppmRegisterConnInputEvent`	input	Bind an output port to an event
`ppmConnGetOutputInfo`	output	Get info about the FIFO on output an port
`ppmConnGetInputInfo`	input	Get info about the FIFO on an input port

9.1 ppmConnPut

Prototype

```
Bool ppmConnPut(ppmConnOutputHandle conn, void *value);
```

Description

Puts the given data into the FIFO if there is space and returns True. If not it returns False. The size of data copied to the FIFO depends on the specified width in bits, rounded up the the nearest byte. The port handle was initialised by the simulator to a non-zero value if the port is connected or to zero if not. It is an error to put data into an unconnected port.

Example

```
#include “peripheral/ppm.h”

// initialised by the simulator (see section 2.5)
ppmConnOutputHandle outputPort;

void putWord (char data) {

    if(outputPort ) {
        if(ppmConnPut(outputPort, &data)) {
            bhmPrintf(“Data was sent\n”);
        } else {
            bhmPrintf(“Data was NOT sent\n”);
        }
    }
}
```

9.2 ppmConnGet

Prototype

```
Bool ppmConnGet(ppmConnInputHandle conn, void *value);
```

Description

Gets data from the FIFO if available and returns True. If not it returns False.
The size of data copied from the FIFO depends on the specified width in bits, rounded up the the nearest byte. The port handle was initialised by the simulator to a non-zero value if the port is connected or to zero if not. It is an error to get data from an unconnected port.

Example

```
#include “peripheral/ppm.h”

// initialised by the simulator (see section 2.5)
ppmConnInputHandle inputPort;

void getData (void) {

    if(inputPort ) {
        char data;
        if(ppmConnGet(inputPort, &data)) {
            bhmPrintf(“Data ‘%c’ was received\n”, data);
        } else {
            bhmPrintf(“Data was NOT received\n”);
        }
    }
}
```

9.3 ppmRegisterConnInputEvent

Prototype

```
Bool ppmRegisterConnInputEvent(ppmConnInputHandle conn, bhmEventHandle event);
```

Description

Binds a FIFO input port to a pre-defined event. The event will be triggered (once) when data becomes available in the FIFO. Returns True if the binding was successful.

Example

```
#include “peripheral/ppm.h”

// initialised by the simulator (see section 2.5)
ppmConnInputHandle inputPort;

void constructor(void) {
    if(inputPort ) {
        bhmEventHandle inputEvent = bhmCreateEvent();
        ppmRegisterConnInputEvent(inputPort, inputEvent);
    }
}

char getWord (void) {

    char data = 0;
    if(inputPort ) {
        while(!ppmConnGet(inputPort, &data)) {
            bhmWaitEvent();
        }
        bhmPrintf(“Data received\n”);
    }
    return data;
}
```

9.4 ppmRegisterConnOutputEvent

Prototype

```
Bool ppmRegisterConnOutputEvent(ppmConnOutputHandle conn, bhmEventHandle event);
```

Description

Binds a FIFO output port to a pre-defined event. The event will be triggered (once) when space for data becomes available in the FIFO. Returns True if the binding was successful.

Example

```
#include “peripheral/ppm.h”

// initialised by the simulator (see section 2.5)
ppmConnInputHandle outputPort;

void constructor(void) {
    if(outputPort) {
        bhmEventHandle outputEvent = bhmCreateEvent();
        ppmRegisterConnOutputEvent(outputPort, outputEvent);
    }
}

char putWord (char data) {

    if(outputPort) {
        while(!ppmConnPut(outputPort, &data)) {
            bhmWaitEvent();
        }
        bhmPrintf(“Data sent\n”);
    }
}
```

9.5 ppmGetConnInputInfo

Prototype

```
Bool ppmGetConnInputInfo (ppmConnInputHandle conn, octcnConnInfoP info);
```

Description

Fetches information from an input port about the connected FIFO.

Example

```
#include “peripheral/ppm.h”
#include “ocl/oclcnTypes.h”

// initialised by the simulator (see section 2.5)
ppmConnInputHandle inputPort;

void printInfo(void) {
    if(inputPort) {
        octcnConnInfo info;

        ppmGetConnInputInfo(inputPort, &info);

        bhmPrintf(“words     : %u\n”, info->words);
        bhmPrintf(“bits      : %u\n”, info->bits);
        bhmPrintf(“numFilled : %u\n”, info->numFilled);
        bhmPrintf(“numEmpty  : %u\n”, info->numEmpty);
    }
}
```

Notes

The structure `octcnConnInfo` is defined in file

`$IMPERAS_HOME/ImpPublic/include/host/ocl/oclcnTypes.h`

This structure is also used by the `vmi` interface.

9.6 ppmGetConnOutputInfo

Prototype

```
Bool ppmGetConnOutputInfo (ppmConnOutputHandle conn, octcnConnInfoP info);
```

Description

Fetches information from an output port about the connected FIFO.

Example

```
#include “peripheral/ppm.h”
#include “ocl/oclcnTypes.h”

// initialised by the simulator (see section 2.5)
ppmConnOutputHandle outputPort;

void printInfo(void) {
    if(outputPort) {
        octcnConnInfo info;

        ppmGetConnOutputInfo(inputPort, &info);

        bhmPrintf(“words     : %u\n”, info->words);
        bhmPrintf(“bits      : %u\n”, info->bits);
        bhmPrintf(“numFilled : %u\n”, info->numFilled);
        bhmPrintf(“numEmpty  : %u\n”, info->numEmpty);
        bhmPrintf(“inputs    : %u\n”, info->inputs);
        bhmPrintf(“outputs   : %u\n”, info->outputs);
    }
}
```

Notes

The structure `octcnConnInfo` is defined in file

`$IMPERAS_HOME/ImpPublic/include/host/ocl/oclcnTypes.h`

This structure is also used by the `vmi` interface.

10 Serial Device Support

This interface provides a serial channel to a device outside the simulation environment. It is intended for use in a serial character device model such as a UART.

Using this interface is optional but will ensure the model has a control interface similar to other serial devices.

There are four functions and one variant:

Function	**Use**
bhmSerOpenAuto	Open a new serial channel using standard model parameters.
bhmSerReadN	Read available characters (does not block).
bhmSerWriteN	Write characters (does not block).
bhmSerReadB	Read available characters (can block).
bhmSerWriteB	Write characters (can block).
bhmSerClose	Close the channel and flush output.
bhmSerOpen	Open a new channel. Does not use standard model parameters.

The definition of a serial device using the model generation tool, iGen, should also make use of the formal macro BHM_SER_OPEN_AUTO_FORMALS. This defines all of the parameters that are automatically added to a model when the serial interface is used.

See the section 10.1 bhmSerOpenAuto for a description of the parameters.

10.1 bhmSerOpenAuto

Prototype

```
Int32 bhmSerOpenAuto (void);
```

Description

Create a new serial channel using parameters specified by the platform.

Returns a positive integer which should be passed as the `channel` argument to other `bhmSer*()` functions. This function cannot fail – the simulator will exit if an error occurs.

Using this function gives the model the following parameters, which can be set in the platform in the usual way:

Parameter	Type	Meaning
client	Boolean	If true (client mode), connect to a listening socket with the `hostname` and `portnum` supplied. If false (server mode), create a listening socket on this host.
hostname	String	In client mode, specify the host to connect to.
nonblocking	Boolean	If true, `bhmSerOpenAuto()` will not wait for a connection, allowing simulation to proceed. Output data will be discarded. No input data will arrive. If false, `bhmSerOpenAuto()` will block until a connection is made.
console	Boolean	If true, an interactive console window will be opened on the host and connected to the port. Parameters `portnum`, `portfile` & `infile` will be ignored.
portnum	Integer	In client mode, specify the port to connect to. In server mode, listen on this TCP/IP port for a connection. If zero, allocate a TCP port from the pool and listen on that port. The allocated port number will be reported on the simulator console. (see `portFile` below).
infile	String	Name of file to read for device input instead of using a port or a console. Note that each call to `bhmSerRead()` will read as many characters as requested from this file.
outfile	String	Name of file to write device output. Can be used in addition to `console` or `portnum`.
portFile	String	In server mode, if `portnum` was specified as zero, write the allocated port number to this file.
log	Boolean	If true, serial output will be reported to the simulator log in addition to other outputs.
finishOnDisconnect	Boolean	If true, disconnecting the port will cause the simulation to finish.
xchars	Integer	When `console` is true these parameters can be used to modify the initial character dimensions of the console.
ychars	Integer	

If none of `console`, `portnum` or `infile` are specified in the platform, calls to the `bhmSerRead` functions will always return 0.

If none of `console`, `portnum` or `outfile` are specified in the platform, calls to the `bhmSerWrite` functions for the channel will always return 0 and data written will be discarded.

Nonblocking mode allows the simulation to continue without a connection. The connection can occur any time a `bhmSerRead` or `bhmSerWrite` function is called. If

`finishOnDisconnect` is false, disconnecting a port (from the other end) does not stop the simulation; a new connection can be made at any time.
Nonblocking mode does not affect the blocking behaviour of the `bhmSerRead` or `bhmSerWrite` functions.

In server mode, the listening port is on the local host. If client mode, `hostname` can specify another host. The local host's name resolution service is used. `portnum` is usually set to a convenient number for connection to another simulation or program. In a scripted environment, `portnum` can be set to zero; the port is chosen by the host so does not clash with other simulations on the same host.

10.2 bhmSerOpen()

Prototype

```
Int32 bhmSerOpen (
    Uns32      *portp,
    const char *logfile,
    const char *outfile,
    const char *sourcefile,
    const char *portfile,
    Bool        client,
    Bool        nonblocking,
    Bool        verbose,
    Bool        console,
    Bool        finishOnDisconnect,
    Uns32       xchars,
    Uns32       ychars
);
```

Description

Create a new serial channel. This function is identical to `bhmSerOpenAuto()`but is configured with arguments instead of model parameters. Please refer to section 10.1. Note that a model using more than one channel with different parameters values must use this function (since model parameters are common to the whole model, and not specific to the channel).

10.3 bhmSerReadN

Prototype

```
Uns32 bhmSerReadN (Int32 channel, Uns8 *buffer, Uns32 bytes);
```

Description

Read as many bytes as are available from the channel, up to the maximum specified by `bytes`, into the supplied buffer and return how many were actually read.

This call will not block. Use function `bhmSerReadB()` instead if blocking semantics are required.

One usage scenario for this function is to call it at the period of the intended baud rate of the device, using `bhmWaitDelay()` to create the interval, as shown in the example below. Note that this will not be the real-time baud-rate.

Example

```
#include "peripheral/bhm.h”

Int32 ch = bhmSerOpenAuto();

while(notDeadYet) {
    bhmWaitDelay(convertToMicroSeconds(getBaudRateReg));
    Uns8 c;
    Int32 actual = bhmSerReadN(ch, &c, 1);
    If(actual) {
        putInRxRegister(c);
        setRxReadyBit();
    }
}
bhmSerClose(ch);
```

10.4 bhmSerWriteN

Prototype

```
Uns32 bhmSerWriteN (Int32 channel, Uns8 *buffer, Uns32 bytes);
```

Description

Attempt to send the given number of bytes from `buffer` to the serial device and return how many were actually sent.

This call will not block. Use function `bhmSerWriteB()` instead if blocking semantics are required.

If the output is being written to a file it will be flushed after the data is written.

Example

```
#include "peripheral/bhm.h”

Int32 ch = bhmSerOpenAuto();

…
    if (txDataReady()) {
        Uns8 c = getTxData();
        Int32 actual = bhmSerWriteN(ch, &c, 1);
        if(actual != 1) {
            errorReport();
        }
    }
…
bhmSerClose(ch);
```

10.5 bhmSerReadB

Prototype

```
Uns32 bhmSerReadB (Int32 channel, Uns8 *buffer, Uns32 bytes);
```

Description

Read as many bytes as are available from the channel, up to the maximum specified by `bytes`, into the supplied buffer and return how many were actually read.

This function will block the current PSE thread until data is available. Use function `bhmSerReadN()` instead if non-blocking semantics are required. Note that only the PSE thread is blocked (not the simulation as a whole).

One usage scenario for this function is to call it at the period of the intended baud rate of the device, using `bhmWaitDelay()` to create the interval, as shown in the example below. Note that this will not be the real-time baud-rate.

Example

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

Int32 ch = bhmSerOpenAuto();

while(notDeadYet) {
    bhmWaitDelay(convertToMicroSeconds(getBaudRateReg));
    Uns8 c;
    Int32 actual = bhmSerReadB(ch, &c, 1);
    If(actual) {
        putInRxRegister(c);
        setRxReadyBit();
    }
}
bhmSerClose(ch);
```

10.6 bhmSerWriteB

Prototype

```
Uns32 bhmSerWriteB (Int32 channel, Uns8 *buffer, Uns32 bytes);
```

Description

Attempt to send the given number of bytes from `buffer` to the serial device and return how many were actually sent.

This function will block the current PSE thread until data can be written. Use function `bhmSerWriteN()` instead if non-blocking semantics are required. Note that only the PSE thread is blocked (not the simulation as a whole).

If the output is being written to a file it will be flushed after the data is written.

Example

```
#include "peripheral/bhm.h”

Int32 ch = bhmSerOpenAuto();

…
    if (txDataReady()) {
        Uns8 c = getTxData();
        Int32 actual = bhmSerWriteB(ch, &c, 1);
        if(actual != 1) {
            errorReport();
        }
    }
…
bhmSerClose(ch);
```

10.7 bhmSerClose

Prototype

```
void bhmSerClose (Int32 channel);
```

Description

Close an open channel, flushing any buffered data.

10.8 bhmSerLastError

Prototype

```
Uns32 bhmSerLastError (Int32 channel);
```

Description

If `bhmSerReadN`, `bhmSerReadB`, `bhmSerWriteN` or `bhmSerWriteB` return zero, use this function to check the unix-like `errno`.

Example

```
#include "peripheral/bhm.h”

Uns8 c;
Int32 actual = bhmSerReadB(ch, &c, 1);
if(actual ==0) {
    Uns32 error = bhmSerLastError(ch);
    if (error != 0) {
        bhmPrintf(“Error %u during bhmSerReadB\n”, error);
    }
}
```

10.9 Record and Replay

The serial channel is subject to the simulator's record and replay feature: in record mode, all serial input data is recorded to a file specified by the simulator. In replay mode, the normal channel input is disabled and replaced with the replay data, such that data is presented at the same rate as in the recording. `bhmSerRead` is a polled interface; calls that return data or no data will occur in the same order as recorded. The serial channel will check for differences in the time of each call.

10.10 Notes and Restrictions

If using more than one channel in a platform, pay attention to which channel is connected to which external device; channels will block for a connection in the order that their models are instanced.

11 Ethernet Device Support

This group of functions connects a model of an ethernet device to either a virtual LAN which uses the host computers TCP/IP socket IP, the physical ethernet of the host computer or to a virtual network modeled using packetnets. Section 11.9 describes the three modes of operation.

Summary:

Function	**Use**
`bhmEthernetOpenAuto`	Connect to ethernet. Configuration is by platform parameters
`bhmEthernetOpen`	Connect to ethernet. Configuration is from function arguments
`bhmEthernetWriteFrameB`	Write a frame to ethernet. Blocking
`bhmEthernetWriteFrameN`	Write a frame to ethernet. Non-blocking.
`bhmEthernetInstallCB`	Specify a function to be called when each packet arrives

The model should open the ethernet channel in its constructor by calling either `bhmEthernetOpenAuto()` or `bhmEthernetOpen()`. The returned handle is passed to the other functions. The model should define a function to process incoming packets and install it using `bhmEthernetInstallCB()`

Note that a peripheral model can have only **one instance** of the ethernet interface.

The write functions have a blocking and non-blocking version. In the blocking version, if the operation would block, then the peripheral is de-scheduled until the end of the quantum, when the operation is re-tried. The non-blocking version will either succeed or fail immediately.

The model should assemble ethernet frames then send them to the virtual network using `bhmEthernetWriteFrameB or bhmEthernetWriteFrameN.`

The channel should be closed during model destruction.

11.1 bhmEthernetOpenAuto

Prototype

```
Int32 bhmEthernetOpenAuto (void);
```

Description

Open a new ethernet channel. The ethernet model can be configured using a standard set of model parameters that will be compatible with other ethernet models. This is the recommended function.

The model must declare these parameters:

Parameter	Purpose
`tapDevice`	Enable TAP mode and name the TAP device. Passing null puts the device in User Mode.
`Redir`	Redirection instruction. For use with User Mode only.
`tftpPrefix`	Enable TFTP server and set the path to the root of the TFTP directory
`network`	Set the User Mode (IPv4) address of the local network device.
`logfile`	Record a `wireshark` compatible log to a file of this parameter's value

The macro `BHM_ETHERNET_OPEN_AUTO_FORMALS` contains declarations of the parameters so should be used as per the example.

Return

Returns a non-negative handle if successful, -1 if the operation fails (the system will print an error message in this case).

Example

To declare parameters for use by `bhmEthernetOpenAuto()`

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

static ppmParameter parameters[] = {

    ....
    BHM_ETHERNET_OPEN_AUTO_FORMALS,
    ....
    { 0 }
};

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

ppmModelAttr modelAttrs = {
    ....
    .paramSpecCB = nextParameter,
    ....
};
```

To initialize the interface:

```
#include “peripheral/bhm.h”

Int32 ethernet;

int main(....) {
    ....
    ethernet = bhmEthernetOpenAuto();
    ....
}
```

Restrictions

A model should not open more than one ethernet channel.

Notes

Wireshark file format is used by tools for analyzing ethernet traffic. It can be read by tools including Wireshark and tcpdump.

11.2 bhmEthernetOpen

```
Prototype Int32 bhmEthernetOpen(
    const char *tap_device,
    const char *redir,
    const char *tftp_path
);
```

Description

Open a new ethernet channel. The ethernet model must be configured using the function arguments. `bhmEthernetOpenAuto()` is usually more convenient to use.

Function arguments:

Argument	Purpose
`tapDevice`	Enable TAP mode and name the TAP device. Passing null puts the device in User Mode.
`redir`	Redirection instruction. For use with User Mode only.
`tftpPrefix`	Enable TFTP server and set the path to the root of the TFTP directory

Return

Returns a zero or positive handle if successful, -1 if the operation fails (the system will print an error message in this case).

Example

To initialize the interface:

```
#include “peripheral/bhm.h”

Int32 ethernet;

int main(....) {
    ....
    ethernet = bhmEthernetOpen(“tap”, NULL, “/var/tmp/imperasFTPdir”);
    ....
}
```

Restrictions

A model should not open more than one ethernet channel.

11.3 bhmEthernetReadFrameB

Prototype

```
Uns32 bhmEthernetReadFrameB(
    Int32 ch,
    Uns8 *buffer,
    Uns32 length,
    Uns32 timeMS,
    Uns32 poll
);
```

Description

(Deprecated. Use `bhmEthernetInstallCB`).

11.4 bhmEthernetReadFrameN

Prototype

```
Uns32 bhmEthernetReadFrameB(
    Int32 ch,
    Uns8 *buffer,
    Uns32 length,
);
```

Description

(Deprecated. Use `bhmEthernetInstallCB`).

11.5 bhmEthernetWriteFrameB

Prototype

```
Uns32 bhmEthernetWriteFrameB(
    Int32 ch,
    Uns8 *buffer,
    Uns32 length,
);
```

Argument	Purpose
`Int32 ch`	Channel number returned by bhmEthernetOpenAuto() etc.
`Uns32 *buffer`	Buffer to be sent
`Uns32 length`	Number of bytes to be sent

Return

Returns the number of characters sent. This will be `length` or zero if the operation fails.

Description

Waits until the frame is transmitted. Returns the length of the transmitted frame or zero if the operation fails. Note that the current PSE thread will be blocked though other threads in this model could run if they are enabled.

11.6 bhmEthernetWriteFrameN

Prototype

```
Uns32 bhmEthernetWriteFrameB(
    Int32 ch,
    Uns8 *buffer,
    Uns32 length,
);
```

Argument	Purpose
`Int32 ch`	Channel number returned by bhmEthernetOpenAuto() etc.
`Uns32 *buffer`	Buffer to be sent
`Uns32 length`	Number of bytes to be sent

Return

Returns the number of characters sent. This will be `length` or zero if the operation fails.

Description

Returns immediately the length of the transmitted frame or zero if the operation fails. The current PSE thread will not be blocked.

11.7 bhmEthernetInstallCB

Prototype

```
typedef void (*bhmEtherPacketFn)(void *userData, Uns8 *data, Uns32 bytes);

void bhmEthernetInstallCB(
   Int32            ch,
   bhmEtherPacketFn cb,
   void            *userData
);
```

Argument	Purpose
`Int32 ch`	Channel number returned by bhmEthernetOpenAuto() etc.
`bhmEtherPacketFn cb`	Function to be called with each packet
`void *userData`	pointer passed to the callback

Description

Install a callback to be called each time a packet arrives from the ethernet interface.

11.8 bhmEthernetClose

Prototype

```
Uns32 bhmEthernetClose (Int32 ch);
```

Description

Closes the given channel.

11.9 Modes

The ethernet interface has these modes:

1. User mode is selected by default.
2. TAP mode is selected by setting the `tapDevice` parameter.
3. Packetnet mode is selected by connecting packetnet port `phy` to at least one other model.

11.10 User Mode

User mode (also known as SLiRP) makes no special requirements on the host; it runs with user privileges, but implements a limited set of networking functions.

Operation

User mode creates a virtual subnet with default addresses 10.0.2.x. Our device has the default IP address 10.0.2.15. SLiRP creates a gateway with address 10.0.2.2 from this subnet to the host's network and performs Network Address Translation.

User mode forwards TCP and UDP requests across the bridge and maintains virtual circuits so that replies are routed back to the interface. Software running on the platform can access the host's network as a client using protocols such as FTP, Telnet and HTTP. By default, incoming requests are blocked so the device cannot operate as a server. However, the `redir` parameter can be used to redirect requests to a particular port number on the host into the model on a different port number.

User mode has these features:

1. A TFTP (Trivial File Transfer Protocol) server.
   A platform can use the model to fetch files from the host's file system using a TFTP client. The TFTP server responds to the TFTP protocol directed to any IP address.

2. A DHCP server.
   The model is granted an IP address from a simulated local network so it does not clash with devices on the host's physical network. Addresses are allocated by default in the 10.0.2.x range.

3. A NAT bridge to the hosts computer's ethernet.
   The model can communicate with the ethernet of the host computer (and hence with the internet).

11.10.1 User Mode Redirection

The redir parameter or argument is a string containing a comma separated list of redirection commands. Each redirection is of the form:

```
<protocol>:<host port>:<ip address>:<virtual port>
```

for example, this will redirect the host's port 4444 to the telnet port of the model, allowing a user to telnet to a guest linux running on a virtual platform.

```
tcp:4444::23
```

11.10.2 Changing the Network address

The network parameter changes the network address from its default of 10.0.0.0. The format is the standard period-separated four numbers 0-255. e.g.

```
network=192.168.0.0
```

Note that the values from this parameter are masked to prevent being set to a public IP address. Run the simulator with `–verbose` to see a summary of IP addresses from each ethernet adapter.

11.11 TAP Mode

TAP mode is selected by setting the `tapDevice` parameter to the name of the TAP device. In this mode a driver is installed on the host computer which makes the host's ethernet device respond to the ethernet address of this model's interface. The model connects (exclusively) to this device and is now visible on the host's ethernet and can perform any function of a real ethernet device. This mode has more capability but requires root access to the host. The `redir` parameter is not used in this mode.

11.11.1 Configuring the host

In order to communicate from the target to host and/or Internet some network configuration may be needed,
both on target and on host. This section lists the necessary steps to get the tunneling working, and
following sections show some example configurations. The exact steps will depend on OS versions and on
the existing kernel and network configurations on host and target.

The first five steps to get tunneling working differ based on the host operating system. The last steps are the same regardless of the host.

11.11.1.1 TAP Setup for Linux Host Platforms

1. Create a TAP device on the host. Ensure it's accessible by the user starting simulation. For example:

```
# create tap0 interface, owned by <user>
host> tunctl -u <user> -t tap0
```

Normally, this loads automatically the tun driver as needed, but it may be necessary to do this manually.

2. Configure host's TAP interface. For example:

```
host> ifconfig tap0 192.168.9.4 up
```

Make sure that target.eth0 and host.tap0 are on the same private network, e.g. 192.168.9.0/24.

3. Ensure connectivity between tap0 and other interfaces on the host. Different approaches are possible, e.g. IP forwarding, bridging, proxy ARP. For IP forwarding:

```
host> sysctl -w net.ipv4.ip_forward=1
```

4. Modify the host's firewall as needed, especially in TAP mode, so modify host forward filtering as needed. For example:

```
host> iptables -t filter -I FORWARD -i tap0 -j ACCEPT
```

In some cases this isn't enough. Inspect the "filter" table with:

```
host> iptables -t filter -L
```

5. In TAP mode, configure SNAT as needed. If the tap0 interface uses a reserved IP address, e.g. `192.168.9.4`, the packets that it sends to Internet must have their source IP translated using SNAT (Source Network Address Translation, or masquerading), otherwise response packets will never reach back to it. Force SNAT if Inet router needs it (where `wlan0` is a host interface connected to Internet). For example:

```
host> iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE   #
```

11.11.1.2 TAP Setup for Windows Host Platforms

1. Download the TAP-windows virtual ethernet adapter from http://openvpn.net/index.php/download/community-downloads.htm

and install it by running the installer as administrator:

- When choosing components, select "TAP Virtual Ethernet Adapter" to install a virtual network interface; and "TAP Utilities" to provides menu commands "TAP-Windows - Utilities - Add a new TAP virtual ethernet adapter" and "TAP-Windows - Utilities - Delete ALL TAP virtual ethernet adapters". that can be run later as the administrator to change things.
- Choose an installation directory. The default is satisfactory.

After installation, "Device Manager" will show an additional entry under "Network adapters" with a name starting with "Tap-Windows Adapter V9". In addition, within the "Control Panel" context "Network and Internet - Network Connections" there will be an additional entry referencing the added network adapter. This entry usually is named something like "Local Area Connection <n>" and has the initial state "Network cable unplugged".

⇒ If you are using OpenVPN for virtual private networking, it seems to use
⇒ the last TAP-windows virtual adapter installed. To verify, disconnect and then
⇒ re-connect with OpenVPN while watching the status of the icons in the
⇒ *Network and Internet - Network Connections*.
⇒ Then, in the next step, choose *TAP_Windows Adapter V9* <n>
⇒ that is shown with the state *Network cable unplugged*.

2. Configure the host's TAP interface. This includes giving the network connection a

name that is easily referenced via a command line argument and IP address. These steps can be done in the GUI from the

Network and Internet - Network Connections:

Select the icon for the connection and rename it to a (preferable short) single word name like `tap0`.

- Assign an IP address by executing "Properties" for the icon, double-clicking on *Internet Protocol Version 4 (TCP/IPv4)*, selecting *Use the following IP address*, and then entering an *IP address* and *Subnet mask*. For example `192.168.9.4` and `255.255.255.0` respectively.

The IP address can also be changed from the command line as administrator by using this command:

```
host> netsh interface ipv4 set address static tap0 192.168.9.4 255.255.255.0
```

3. Ensure connectivity between the virtual ethernet address and other interfaces on the host. This is done by enabling IP routing on the Windows host. For example, run services.msc and ensure that "Routing and Remote Access" service is enabled and either started or set to automatically start.

The command prompt can be used to verify that IP routing is enabled, e.g.

```
host> ipconfig /all
```

4. Modify host's firewall as needed, especially in TAP mode.

For example, it may be necessary to enable ICMP requests in order to ping host.tap0 from target.eth0. For example:

```
host> netsh firewall set icmpsetting 8 enable
```

5. In TAP mode, configure Internet Connection Sharing. For example, on Vista run services.msc and enable "Internet Connection Sharing", then edit the properties of the Internet network connection to enable sharing with tap0.

NOTE: this may change the manually assigned IP to tap0 in step 2 above, and the following examples need to be tweaked accordingly.

6. Temporarily disable anti-virus software or configure the software to make an exception for the platform executable - platform.exe. This is needed as the accesses to the host resources by the virtual platform may be mistaken as those made by code that the anti-virus software is designed to thwart.

11.11.1.3 Common Steps for TAP Setup

1. Find out the dwc_emac instances that will be used by target kernel. For example, the AlteraCycloneV platform has two instances of the dwc_emac model: iEMAC0 and iEMAC1. The default socfpga configuration in vanilla Linux kernels enables only the second instance, iEMAC1.

2. Start simulation, using the desired network mode for each dwc_emac instance. To start AlteraCycloneV with iEMAC0 in SLiRP (default) mode and iEMAC1 in TAP mode:

```
host> platform.exe --override AlteraCycloneV/iEMAC1/tapDevice=tap0
```

3. Verify that the target OS has loaded the corresponding NIC driver and it has successfully initialized the model. To check this on Linux:

```
target> ifconfig -a
```

The default socfpga configuration in vanilla Linux kernels has the stmicro driver linked into the kernel. On other setups it may be necessary to manually load the driver.

4. Configure the network interface, e.g. eth0, on target, if the previous step shows that it isn't configured already. On a Linux target this can be done with:

```
target> ifconfig eth0 10.0.2.7 netmask 255.255.255.0 up
```

In SLiRP mode, target.eth0 interface and slirp.gw interface have to be on the same network, which is usually 10.0.2.0/24.

In TAP mode, target.eth0 and host.tap0 interfaces have to be on the same network. Following examples use 192.168.9.0/24.

5. Update target's IP routing. After configuring the eth0 interface on the previous step, the target should know how to communicate with nodes from eth0's network (e.g. 10.0.2 or 192.168.9). On Linux this can be verified with:

```
target> route -n           # check routing to eth0's network
```

To update the routing on a Linux target for communication with nodes on other networks:

```
target> route add default gw 10.0.2.2                     # add default GW
```

6. Update target's DNS resolution. On a Linux target:

```
# (SLiRP mode)
target> echo "nameserver 10.0.2.3" > /etc/resolv.conf

# TAP mode
target> echo "nameserver 192.168.0.1" > /etc/resolv.conf
```

11.11.2 Example Uses

11.11.2.1 Manual Target Network Configuration

```
# start simulation
host> platform.exe
```

```
# start and configure eth0
target> ifconfig eth0 10.0.2.7 netmask 255.255.255.0 up

# add default GW
target> route add default gw 10.0.2.2

# set up DNS
target> echo "nameserver 10.0.2.3" > /etc/resolv.conf

# browse google
target> lynx http://www.google.com/
```

11.11.2.2 DHCP Target Network Configuration

```
# start simulation
host> platform.exe

# start eth0
target> ifconfig eth0 up

# lease an IP address from DHCP server
target> udhcpc -i eth0

# configure eth0
target> ifconfig eth0 10.0.2.15 netmask 255.255.255.0

# add default GW
target> route add default gw 10.0.2.2

# set up DNS
target> echo "nameserver 10.0.2.3" > /etc/resolv.conf

# browse google
target> lynx http://www.google.com/
```

11.11.2.3 Linux Host TAP Networking Configuration

with IP Routing (requires root access)

target.eth0 / host.tap0 connectivity:

```
# create tap0 interface, owned by <user>
host> tunctl -u <user> -t tap0

# start tap0
host> ifconfig tap0 192.168.9.4 up

# start simulation
host> platform.exe --override AlteraCycloneV/iEMAC1/tapDevice=tap0

# start eth0
target> ifconfig eth0 192.168.9.3 up

# check connection
target> ping -c 1 192.168.9.4
```

host.tap0 / Internet connectivity, using IP forwarding and NAT:
(this example assumes that host has a wlan0 interface with a reserved address, e.g. 192.168.0.5, connected to an Internet router 192.168.0.1, which provides also DNS):

```
# enable IP forwarding
```

```
host> sysctl -w net.ipv4.ip_forward=1

# force SNAT if Inet router needs it
host> iptables -t nat -A POSTROUTING -o wlan0 -j MASQUERADE

# modify host forward filtering as needed
host> iptables -t filter -I FORWARD -i tap0 -j ACCEPT

# add default GW
target> route add default gw 192.168.9.4

# set up DNS
target> echo "nameserver 192.168.0.1" > /etc/resolv.conf

# browse Google
target> lynx http://www.google.com/
```

```
           target              host           Internet router
              |                 / \                  |
192.168.9.3|   192.168.9.4/   \192.168.0.5   |192.168.0.1
           eth0            tap0   wlan0                /
              \____________/       \____________/
```

11.11.2.4 Linux Host TAP Networking Configuration

Bridging Two Targets (requires root access)
target1.eth0 / target2.eth0 / host.br0 connectivity

```
# create tap interfaces, owned by <user>
host> tunctl -u <user> -t tap0
host> tunctl -u <user> -t tap1

# ensure tap interfaces are un-configured
host> ifconfig tap0 0.0.0.0 down
host> ifconfig tap1 0.0.0.0 down

# create a bridge interface
host> brctl addbr br0
host> brctl addif br0 tap0

# attach tap interfaces to the bridge
host> brctl addif br0 tap0
host> brctl addif br0 tap1

# activate bridge and its ports
host> ifconfig tap0 up
host> ifconfig tap1 up
host> ifconfig br0 up
    or
# bridge may be assigned an IP address to communicate with host
host> ifconfig br0 192.168.9.1 up

# check that filtering doesn't limit bridge's functionality
host> ebtables -t filter –L

# start two simulators
host> platform.exe --override AlteraCycloneV/iEMAC1/tapDevice=tap0
host> platform.exe --override AlteraCycloneV/iEMAC1/tapDevice=tap1

# start eth0 on each target
target1> ifconfig eth0 192.168.9.3 up
target2> ifconfig eth0 192.168.9.4 up

# check the connections
```

```
target1> ping -c 1 192.168.9.1
target2> ping -c 1 192.168.9.3
```

```
target1              host                 target2
  |                   |[192.168.9.1]         |
  |                   br0                    |
  |192.168.9.3   port1/   \port2             |192.168.9.4
 eth0            tap0       tap1            eth0
   \______________/          \____________/
```

11.11.2.5 Windows Host TAP Networking Configuration

With IP Routing (requires administrator access)
target.eth0 / host.tap0 connectivity

```
# ensure Tap-windows has been installed and named "tap0"
host>   netsh interface ipv4 set address tap0 static 192.168.9.4 255.255.255.0
host>   netsh interface ipv4 set interface tap0 Forwarding=enabled

# start simulation
host>   platform.exe --override AlteraCycloneV/iEMAC1/tapDevice=tap0

# start eth0 interface
target> ifconfig eth0 192.168.9.3 up
target> ping -c 1 192.168.9.4

# check the connection
host>   ping 192.168.9.3
```

11.12 Packetnet mode

A peripheral model with an ethernet interface should have a packetnet port called phy. Connecting this port to other packetnet devices will enable packetnet mode. When the device uses the `bhmEthernet` API to send a packet, it will be distributed to other models on the packetnet (see section 11.5). When other models send a packet, this model's installed callback will be called (see section 11.7).

It is expected that packetnet mode will be used to model a virtual network within the platform. Note that while the `bhmEthernet` API is similar to the `ppmPacketnet` API, writing the model to use the `bhmEthernet` API allows the model to be used unmodified with internal or external networks, whereas using the `ppmPacketnet` API limits the model to using internal networks.

12 USB Device Support

This interface provides a USB channel to a device outside the simulation environment
This interface is under development and cannot be used. Please contact Imperas.

Function	**Use**
`bhmUSBOpen`	Open a new USB interface
`bhmUSBControlTransfer`	Send a control message
`bhmUSBBulkTransfer`	Send bulk data
`bhmUSBClose`	close the interface

12.1 bhmUSBOpen

Prototype

```
Int32 bhmUSBOpen (void);
```

Description

This interface is under development and cannot be used. Please contact Imperas.

12.2 bhmUSBControlTransfer

Prototype

```
Int32 bhmUSBControlTransfer (void);
```

Description

This interface is under development and cannot be used. Please contact Imperas.

12.3 bhmUSBBulkTransfer

Prototype

```
Int32 bhmUSBBulkTransfer (void);
```

Description

This interface is under development and cannot be used. Please contact Imperas.

12.4 bhmUSBClose

Prototype

```
void bhmUSBclose (void);
```

Description

This interface is under development and cannot be used. Please contact Imperas.

13 View Object Interface

This section describes functions to create and provide view objects.

13.1 ppmAddViewObject

Prototype

```
ppmViewObjectP ppmAddViewObject(
    ppmViewObjectP parent,
    const char    *name,
    const char    *description
);
```

Description

Create a view object.

`parent` is a pointer to the parent object (NULL for top level, i.e. peripheral instance).

`description` may be 0.

Notes and Restrictions

None

13.2 ppmSetViewObjectConstValue

Prototype

```
void ppmSetViewObjectConstValue(
    ppmViewObjectP   object,
    vmiViewValueType type,
    void             *pValue
);
```

Description

Set constant value for view object (value copied at time of call).

`pValue` is a pointer to item.

Notes and Restrictions

None

13.3 ppmSetViewObjectRefValue

Prototype

```
void ppmSetViewObjectRefValue(
    ppmViewObjectP   object,
    vmiViewValueType type,
    void             *pValue
);
```

Description

Set value pointer for view object (pointer dereferenced each time value is viewed). Use this to associate a view object with a C variable in the model such that the variable is automatically read when the view object is evaluated.

`pValue` is a pointer to item in persistent memory (must be valid for lifetime of object)

Notes and Restrictions

None

13.4 ppmSetViewObjectValueCallback

Prototype

```
void ppmSetViewObjectValueCallback(
    ppmViewObject      object,
    ppmCBViewValueFunc valueCB,
    void              *userData
);
```

Description

Set value callback for view object.

`valueCB` will be passed the `userData` value and should be declared as:

```
ppmViewValueType valueCB (
    void  *userData,
    void  *buffer,
    Uns32 *bufferSize
) {
 ...
}
```

See the documentation for the vmiviewGetViewObjectValue function the VMI View Function Reference Manual for more info on what this function is expected to return.

Notes and Restrictions

None

13.5 ppmAddViewAction

Prototype

```
void ppmAddViewAction(
    ppmViewObject       object,
    const char         *name,
    const char         *description,
    ppmCBViewActionFunc actionCB,
    void               *userData
);
```

Description

Add an action to a view object.

`actionCB` will be passed the `userData` value and should be declared as:

```
  void actionCB(void * userData);
```

`object` may be 0 for top level, i.e. peripheral instance.

`description` may be 0.

Example

```
#include “peripheral/ppm.h”


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

...

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

Notes and Restrictions

None

13.6 ppmAddViewEvent

Prototype

```
ppmViewEvent ppmAddViewEvent(
    ppmViewObject  object,
    const char    *name,
    const char    *description
);
```

Description

Add an event to a view object

`object` may be 0 for top level, i.e. peripheral instance.

`description` may be 0.

Notes and Restrictions

TBD

13.7 ppmNextViewEvent

Prototype

```
ppmViewEvent ppmNextViewEvent(
    ppmViewObject object,
    ppmViewEvent  old
);
```

Description

Iterate through the view events on a view object.

`old` should be set to 0 for the first call, then the returned value used for each subsequent call until 0 is returned.

`object` may be 0 for top level, i.e. peripheral instance.

Example

```
#include “peripheral/ppm.h”

...
  ppmViewEventP v = NULL;
  while ((v = ppmNextViewEvent(object, v))) {
      // use v here
  }
...
```

Notes and Restrictions

None.

13.8 ppmTriggerViewEvent

Prototype

```
void ppmTriggerViewEvent(ppmViewEvent event);
```

Description

Trigger a view event.

Notes and Restrictions

Noned

13.9 ppmDeleteViewObject

Prototype

```
void ppmDeleteViewObject(ppmViewObject object);
```

Description

Delete a view object (including any child objects).

Notes and Restrictions

None

14 Documentation Interface

The Imperas documentation generator can product a document from a peripheral model. While some parts of the model can be documented automatically by extracting information from the model, some words must be supplied by the model writer to fully describe it. This API is used to produce headings, paragraphs and specific documentation formats for registers or instructions.

The `ppmModelAttr` structure has an entry `docCB` which should be set to a function defined using the `PPM_DOC_FN` prototype. This function should use the following PPM functions to create documentation for the model.

Example

This example is abridged from the OVP arm.ovpworld.org PL011 model.

```
// Define the documentation constructor

ppmDocNodeP Root1_node = ppmDocAddSection(0, "Root");
    {
        ppmDocNodeP doc2_node = ppmDocAddSection(Root1_node, "Description");
        ppmDocAddText(doc2_node, "ARM PL011 UART");
        ppmDocNodeP doc_12_node = ppmDocAddSection(Root1_node, "Limitations");
        ppmDocAddText(doc_12_node, "This is not a complete model of the PL011.");
        ppmDocNodeP doc_22_node = ppmDocAddSection(Root1_node, "Reference");
        ppmDocAddText(doc_22_node, "ARM PrimeCell UART (PL011) (ARM DDI 0183)");
        ppmDocNodeP doc_32_node = ppmDocAddSection(Root1_node, "Licensing");
        ppmDocAddText(doc_32_node, "Open Source Apache 2.0");
    }
    ppmDocNodeP Registers1_node = ppmDocAddSection(0, "Registers");
    {
        ppmDocNodeP dr2_node = ppmDocAddFields(Registers1_node, "dr", 32);
        ppmDocAddText(dr2_node, "UARTDR");
    }
    {
        ppmDocNodeP flags2_node = ppmDocAddFields(Registers1_node, "flags", 32);
        ppmDocAddText(flags2_node, "UARTFR");
            ppmDocAddField(flags2_node, "TXFE", 7, 1);
            ppmDocAddField(flags2_node, "RXFF", 6, 1);
            ppmDocAddField(flags2_node, "TXFF", 5, 1);
            ppmDocAddField(flags2_node, "RXFE", 4, 1);
    }
}

ppmModelAttr modelAttrs = {

    . . . lines omitted for clarity . . .

    .docCB = installDocs,

    . . . lines omitted for clarity . . .
};
```


14.1 ppmDocAddSection

Prototype

```
ppmDocNodeP ppmDocAddSection(ppmDocNodeP parent, const char *name);
```

Description

This function is called to create a new section node. The name of the section is given by the `name` argument. The section could be a root section (in which case the `parent` is `NULL`) or a child of a previously-defined section (given as the `parent` argument).

Once created, a section node will typically be used as a parent of other section nodes or text nodes.

Example

Please refer to the example at the start of this section.

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

14.2 ppmDocAddText

Prototype

```
ppmDocNodeP ppmDocAddText(ppmDocNodeP node, const char *text);
```

Description

This function is called to create a new text node, defining a paragraph with the given text. Leaf-level sections will typically contain one or more text nodes.

Example

Please refer to the example at the start of this section.

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

14.3 ppmDocAddFields

Prototype

```
ppmDocNodeP ppmDocAddFields(
    ppmDocNodeP parent,
    const char *name,
    Uns32       width
);
```

Description

This function is called to create a new documentation node, defining a collection of fields that make up an instruction format. The `width` parameter should specify the overall width of the instruction, usually in bits.

Example

Please refer to the example at the start of this section.

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

14.4 ppmDocAddField

Prototype

```
ppmDocNodeP ppmDocAddField(
    ppmDocNodeP parent,
    const char *name,
    Uns32       offset,
    Uns32       width
);
```

Description

This function is called to create a new documentation node, defining a field within a collection of fields that make up an instruction format. The `width` parameter should specify the width of the field, usually in bits. The `offset` is specified in bits from the least significant end. The parent node must be created with `ppmDocAddFields.`

Example

Please refer to the example at the start of this section.

Notes and Restrictions

1. Fields are numbered from zero, zero being the LSB.
2. Fields can be added to a fields collection in any order.
3. Fields but must not overlap other fields.
4. The most significant bit of the most significant field must not exceed the width of the fields collection.

QuantumLeap Semantics

Synchronizing

14.5 ppmDocAddConstField

Prototype

```
ppmDocNodeP ppmDocAddConstField(
    ppmDocNodeP parent,
    Uns64       value,
    Uns32       offset,
    Uns32       width
);
```

Description

This function is called to create a new documentation node, defining a field within a collection of fields that has a constant value. The `width` parameter should specify the width of the field, usually in bits. The parent node must be created with `ppmDocAddFields`.

Example

Please refer to the example at the start of this section.

Notes and Restrictions

Each field can be added to a fields collection in any order but must not overlap with other fields. The most significant bit of the most significant field must be less than the width of the fields collection. The `value` should not be larger than the specified number of bits can represent.

QuantumLeap Semantics

Synchronizing

14.6 Describing a field

Note that the node returned by `ppmDocAddField` (and `ppmDocAddConstField`) can have further text nodes attached to describe it.

Example

```
VMI_DOC_FN(modelDoc) {

    // Create a new chapter
    ppmDocNodeP chapter = ppmDocAddSection(0, "Added instructions");

    // Document an instruction
    ppmDocNodeP INST1 = ppmDocAddFields(chapter, "INST1", 16);

        // Add fields to the instruction
        ppmDocAddField(INST1, "FIELD1",   0, 4);
        ppmDocAddField(INST1, "FIELD2",   4, 4);
        ppmDocNodeP FIELD3 = ppmDocAddField(INST1, “FIELD3",   8, 4);
        ppmDocAddConstField(INST1,    1,  12,4);

        ppmDocAddText(FIELD3, “This is field3.”);

      . . . more instructions, omitted for clarity . . .
}
```

Used with the Imperas documentation generator, this will produce this documentation:

Chapter 1. Added Instructions

1.1 INST1

15 12	*11 8*	*7 4*	*3 0*
0x1	*FIELD3*	*FIELD2*	*FIELD1*

1.1.1 FIELD3
This is field3.

14.7 ppmDocAddFieldMSLS

Prototype

```
ppmDocAddFieldMSLS(
    parent,
    name,
    msb,
    lsb
)
```

Description

A macro to document a field specifying `MSB` and `LSB` instead of `offset` and `width`.

14.8 ppmDocAddConstFieldMSLS

Prototype

```
ppmDocAddConstFieldMSLS(
    parent,
    value,
    msb,
    lsb
)
```

Description

A macro to create a constant field specifying `MSB` and `LSB` instead `of offset` and `width`.

##