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:	2.2
Filename:	OVP_BHM_PPM_Function_Reference.doc
Last Saved:	Tuesday, 04 August 2015

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

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

    ppmBusPortSpecFn       busPortsCB;       // next bus port callback
    ppmNetPortSpecFn       netPortsCB;       // next net port callback
    ppmPacketnetPortSpecFn packetnetPortsCB; // next net port callback
    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` 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).

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

```
// example


static PPM_PACKETNET_PORT_FN(nextPacketnetPort) {

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

2.5 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`	
`type specifications`	`u`	union of possible type specifications

Each parameter type has a specification structure in a union which can be optionally set to check a parameter's value.

Parameter types:

`ppmParameterType`	**Description**	**Type specification**
`ppm_PT_BOOL`	boolean	specify the 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_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.

Example

```
// Example:

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

2.6 Complete Example:

This is extracted from

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

In this example, the bus net and parameter 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[] = {
    { 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"
    },
};
```

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          // address of the TOP of the stack
);
```

Description

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

A thread requires a stack which must be allocated by the user. Note that the address of the TOP of the stack is passed to `bhmCreateThread`().

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;  // only required if you wish to delete the thread

#define size (64*1024)

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

void myThread(void *user)
{
    Char *string = user;
    while(1) {
   bhmWaitDelay(1000*1000);
        bhmPrintf(“%s\n”, string);
    }
}
```

```
void userInit(void)
{
    struct myThreadContext contextA;
    struct myThreadContext contextB;

    thA = bhmCreateThread(myThread, ”tick”, “threadA”, &stackA[size]);
    thB = bhmCreateThread(myThread, ”tock”, “threadB”, &stackB[size]);
}
```

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).

3.2 *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 user 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.

3.3 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.

3.4 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.

3.5 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.

3.6 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

1. This function should not be called from a net or bus callback (see PPM functions).

3.7 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.

3.8 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.

3.9 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. Should not be called from within a bus or net callback.

3.10 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

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

3.11 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()`.

3.12 bhmGetDiagnosticLevel

Superseded by bhmSetDiagnosticCB().

3.13 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.

3.14 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

None.

3.15 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.

3.16 bhmStringAttribute

Prototype

```
Uns32 bhmStringAttribute(const char *name, char *string, Uns32 length);
```

Description

Read the value of a model string attribute for this model instance. If the attribute is found, its value is placed in the memory pointed to by `string`, up to a maximum of `length` characters and the call returns 1, otherwise the call returns 0.

Example

```
#include "peripheral/bhm.h”

{
    char buff[32];
    if ( !bhmStringAttribute("vendor", &buff, sizeof(buff)) ) {
        bhmMessage("E", "MY_PERIPHERAL", "Missing vendor string attribute");
    }
}
```

Notes and Restrictions

A call to bhmStringAttribute (or bhmIntegerAttribute) searches first in the model definition, then in the model instance and sequentially up the platform hierarchy for an attribute of the given name.

3.17 bhmIntegerAttribute

Prototype

```
Uns32 bhmIntegerAttribute(const char *name, Uns32 *ptr);
```

Description

Read the value of a model integer attribute for this model instance. If the attribute is found, its value is placed in the memory pointed to by `ptr` and the call returns 1, otherwise the call returns 0.

Example

```
#include "peripheral/bhm.h”

{
    Uns32 chipId;
    if ( !bhmIntegerAttribute("chipID", &chipId) ) {
        bhmMessage("E", "MY_PERIPHERAL", "Missing chipID attribute");
    }
}
```

Notes and Restrictions

A call to bhmIntegerAttribute (or bhmStringAttribute) searches first in the model definition, then in the model instance and sequentially up the platform hierarchy for an attribute of the given name. See bhmUns64Attribute for 64-bit numbers.

3.18 bhmUns64Attribute

Prototype

```
Uns32 bhmUns64Attribute(const char *name, Uns64 *ptr);
```

Description

Read the value of a model integer attribute for this model instance. If the attribute is found, its value is placed in the memory pointed to by `ptr` and the call returns 1, otherwise the call returns 0.

Example

```
#include "peripheral/bhm.h”

{
    Uns64 longId;
    if ( !bhmUns64Attribute("longID", &longId) ) {
        bhmMessage("E", "MY_PERIPHERAL", "Missing longID attribute");
    }
}
```

Notes and Restrictions

A call to bhmUns64Attribute searches first in the model definition, then in the model instance and sequentially up the platform hierarchy for an attribute of the given name.

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 attributes "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 "icm/icmCpuManager.h"

icmAttrListP atts = icmNewAttrList();

if(replaymode) {
    icmAddStringAttr(atts, "replay", getMyReplayFile());
}

icmPseP myPSE = icmNewPSE("pse1", psePath, atts, 0, 0);
```

If you are using the Imperas simulator, recording is turned on from the command line:

```
cmd> imperas.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> imperas.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 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.

5.7 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..

5.8 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 ppmCreateRegister 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`

5.9 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.

5.10 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

5.11 ppmInstallReadCallback

Prototype

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

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

Description

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. Only one read callback should be installed on a region. Installing the value NULL removes the callback.
3. The callback should not call bhmWaitEvent() or bhmWaitDelay().
4. Use the prototype macro to declare the callback.

5.12 ppmInstallWriteCallback

Prototype

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

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

Description

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. Only one write callback can be installed on a region. Subsequent calls will replace the original callback. Installing the value NULL removes the last callback.
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.

5.13 ppmInstallChangeCallback

Prototype

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

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

Description

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

6. If the callback writes to it’s own sensitized region, a fatal recursion will occur.
7. Only one write callback can be installed on a region. Subsequent calls will replace the original callback. Installing the value NULL removes the last callback.
8. The callback should not call bhmWaitEvent() or bhmWaitDelay().
9. Use the prototype macro to declare the callback.
10. The model writer might choose to inhibit side effects if the access is a simulation artifact.

5.14 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);
}
```

5.15 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.

5.16 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.

5.17 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 to declare the callback.

5.18 *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.

5.19 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.

5.20 *ppmCreateDynamicSlavePort*

Prototype

```
void ppmCreateDynamicSlavePort(
    const char *slavePort,
    SimAddr    slavePortLoAddress,
    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];   // 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
);
```

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.

5.21 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

```
void *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

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 register",         // 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 register",         // 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.

6.2 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 3 callbacks.
);
```

Description

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
    );
}
```

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 ICM 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 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 attributes.
bhmSerRead	Read available characters (does not block).
bhmSerWrite	Write characters (could block).
bhmSerClose	Close the channel and flush output.
bhmSerOpenBlocking	Open a new channel. Does not use standard model attributes.

9.1 bhmSerOpenAuto

Prototype

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

Description

Create a new serial channel using attributes 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 attributes, which can be set in the platform in the usual way:

Attribute	Type	Meaning
console	Boolean	If set to non-zero an interactive console window will be automatically opened on the host. This is the highest priority attribute, meaning lower priority attributes will be ignored if it is set to non-zero.
portnum	integer	If set, listen on this TCP/IP port for a connection. If set to zero, allocate a TCP port from the pool and listen on that port. The simulation will block until a program (e.g. telnet) connects to this port. The allocated port number will be reported on the simulator console. The portFile attribute may also be used to allow a program to find out the allocated port number. This is the 2nd highest priority attribute.
infile	string	Name of file to read for device input. Each call to bhmSerRead will read as many characters as requested from this file. This is the 3rd highest priority attribute.
outfile	string	Name of file to write device output. Can be used in addition to console or portnum.
portFile	string	If portnum was specified as zero, write the allocated port number to this file when it's known.
log	Boolean	If set to non-zero, serial output will be reported to the simulator log in addition to other outputs.
finishOnDisconnect	Boolean	If set to non-zero, disconnecting the port will cause the simulation to finish.

If none of console, portnum or infile are specified in the platform, calls to bhmSerRead() for the channel will always return 0.

If neither portnum nor outfile are specified in the platform, calls to bhmSerWrite() for the channel will always return 0 and the write data will be ignored.

9.2 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);
```

9.3 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);
```

9.4 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);
```

9.5 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);
```

9.6 bhmSerClose

Prototype

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

Description

Close an open channel, flushing any buffered data.

9.7 bhmSerOpenBlocking

Prototype

```
Int32 bhmSerOpenBlocking (
    Uns32 *portp,
    const char *logfile,
    const char *sourcefile,
    const char *portfile,
    Bool verbose,
    Bool console,
    Bool finishOnDisconnect
);
```

Description

Create a new serial channel using attributes passed in as arguments from the PSE model.

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.

Argument	Meaning
Bool console	If True an interactive console window will be automatically opened on the host. This is the highest priority argument, meaning lower priority arguments will be ignored if it is set to True.
Uns32 *portp	No action if NULL. If not NULL: If *portp!=0, listen on that TCP/IP port for a connection. If *portp==zero, allocate a TCP port from the pool and listen on that port, setting *portp to the allocated port number. The simulation will block until a program (e.g. telnet) connects to this port. The allocated port number will be reported on the simulator console. The portFile attribute may be used when *portp==0 to allow a program to find out the allocated port number. This is the 2nd highest priority argument.
const char *sourcefile	Name of file to read for device input. Each call to bhmSerRead will read as many characters as requested from this file. This is the 3rd highest priority attribute.
const char *logfile	Name of file to write device output. Can be used in addition to other modes.
const char *portfile	If *port==0, write the allocated port number to this file when it's known.
Bool verbose	If True, serial output will go to the simulator log in addition to other outputs.
Bool finishOnDisconnect	If set to non-zero, disconnecting the port will cause the simulation to finish.

9.8 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.

9.9 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.

10 Ethernet Device Support

This interface provides an Ethernet channel to a device outside the simulation environment

There are four functions and one variant:

Function	**Use**
bhmEthernetOpenAuto	TBD
bhmEthernetOpen	TBD
bhmEthernetReadFrame	TBD
bhmEthernetWriteFrame	TBD
bhmEthernetGetStatus	TBD
bhmEthernetClose	TBD

10.1 bhmEthernetOpenAuto

Prototype

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

Description

TBD

10.2 bhmEthernetOpen

Prototype

```
Int32 bhmEthernetOpen (void);
```

Description

TBD

10.3 bhmEthernetReadFrame

Prototype

```
Uns32 bhmEthernetReadFrame (void);
```

Description

TBD

10.4 bhmEthernetWriteFrame

Prototype

```
Uns32 bhmEthernetWriteFrame (void);
```

Description

TBD

10.5 bhmEthernetGetStatus

Prototype

```
Uns32 bhmEthernetGetStatus (void);
```

Description

TBD

10.6 bhmEthernetClose

Prototype

```
Uns32 bhmEthernetClose (void);
```

Description

TBD

11 USB Device Support

This interface provides a USB channel to a device outside the simulation environment

There are four functions and one variant:

Function	**Use**
bhmUSBOpen	TBD
bhmUSBControlTransfer	TBD
bhmUSBBulkTransfer	TBD
bhmUSBClose	TBD

11.1 bhmUSBOpen

Prototype

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

Description

TBD

11.2 bhmUSBControlTransfer

Prototype

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

Description

TBD

11.3 bhmUSBBulkTransfer

Prototype

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

Description

TBD

11.4 *bhmUSBClose*

Prototype

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

Description

TBD

12 View Object Interface

This section describes functions to create and provide view objects.

12.1 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.

Example

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

TBD
```

Notes and Restrictions

TBD

12.2 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.

Example

```
#include “peripheral/ppm.h”

TBD
```

Notes and Restrictions

TBD

12.3 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)

Example

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

TBD
```

Notes and Restrictions

TBD

12.4 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.

Example

```
#include “peripheral/ppm.h”

TBD
```

Notes and Restrictions

TBD

12.5 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

TBD

12.6 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.

Example

```
#include “peripheral/ppm.h”

TBD
```

Notes and Restrictions

TBD

12.7 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.

12.8 *ppmTriggerViewEvent*

Prototype

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

Description

Trigger a view event.

Example

```
#include “peripheral/ppm.h”

TBD
```

Notes and Restrictions

TBD

12.9 ppmDeleteViewObject

Prototype

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

Description

Delete a view object (including any child objects).

Example

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

TBD
```

Notes and Restrictions

TBD

##