

VMM Hardware Abstraction Layer User Guide

HAL Version 2.1
December 2009



Copyright Notice and Proprietary Information

Copyright © 2009 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____.”

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia, Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

1 Introduction

Target Platform Selection	1-3
Hardware-assisted Portion	1-3
Simulated Portion	1-4

2 Implementing Hardware-Assisted Testbenches

Messages	2-2
Hardware-Assisted Layer	2-7
Top-level Module	2-8
Synthesizable Transactors	2-9
Simulated Portion	2-12

A Class Library Reference

vmm_hw_in_if	A-3
vmm_hw_out_if	A-6
vmm_hw_clock	A-9
vmm_hw_clock_control	A-11
OpenVera Interface Bindings	A-14
vmm_hw_in_if_bind	A-15
vmm_hw_out_if_bind	A-17
vmm_hw_clock_bind	A-19

vmm_hw_clock_control_bind	A-20
vmm_hw_arch	A-21
vmm_hw_arch::clk_control().	A-22
vmm_hw_arch::create_in_port().	A-24
vmm_hw_arch::create_out_port().	A-26
vmm_hw_arch::init_sim()	A-28
vmm_hw_arch_null	A-29
vmm_hw_arch_zebu	A-31
vmm_hw_in_port	A-33
vmm_hw_in_port::is_rdy()	A-34
vmm_hw_in_port::wait_is_rdy()	A-35
vmm_hw_in_port::send()	A-37
vmm_hw_out_port	A-38
vmm_hw_out_port::is_rdy()	A-39
vmm_hw_out_port::wait_is_rdy()	A-40
vmm_hw_out_port::receive()	A-41

B Platform Integration

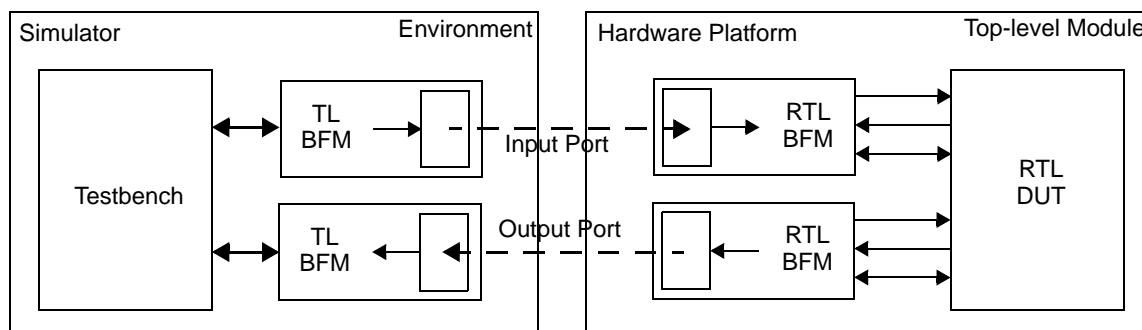
vmm_hw_arch_XYZ	B-3
vmm_hw_arch_XYZ::clk_control()	B-4
vmm_hw_arch_XYZ::create_in_port()	B-7
vmm_hw_arch_XYZ::create_out_port()	B-10
vmm_hw_arch::connect_to()	B-13
vmm_hw_arch::init_sim()	B-15
vmm_hw_in_port_XYZ	B-17
vmm_hw_in_port_XYZ::is_rdy().	B-19
vmm_hw_in_port_XYZ::wait_is_rdy().	B-21
vmm_hw_in_port_XYZ::send().	B-23
vmm_hw_out_port_XYZ	B-25
vmm_hw_out_port_XYZ::is_rdy().	B-27
vmm_hw_out_port_XYZ::wait_is_rdy().	B-29
vmm_hw_out_port_XYZ::receive().	B-31

1

Introduction

The VMM Hardware Abstraction Layer is a VMM application package used to abstract the communication between a testbench running on VCS and a design under verification (DUT) running on an emulator, accelerator or FPGA board, as illustrated in [Figure 1-1](#).

Figure 1-1 Communication Between Testbench and Hardware Assisted



The hardware abstraction layer removes the testbench from the different communication mechanisms provided by various available hardware assistance platforms. The same testbench, testcases and DUT can thus be targetted to different assistance platforms without any modifications.

The application package also contains a purely simulated implementation of the hardware abstraction layer that allows the testbench and testcases to be developed and the DUT to be debugged entirely within the same simulation without requiring any modifications.

The library is accessed by including the file `vmm_hw.sv` or `vmm_hw.vrh` in the testbench source code and by including the file `vmm_hw_rtl.sv` in the synthesizable portion of the testbench source code.

Example 1-1 Accessing the Hardware Abstraction Layer

```
`include "vmm_hw_rtl.sv"
```

Target Platform Selection

The target hardware-assistance platform is selected through a compile-time symbol definition, as defined in [Table 1-1](#). Only one target architecture may be selected at a given time. The same target architecture must be selected for the hardware-assisted portion and the simulated portion.

Table 1-1 Target Architecture Symbols

VMM_HW_ARCH_NULL	Simulation only. No hardware-assistance platform is used.
VMM_HW_ARCH_ZEBU	ZeBu & zTide platforms from EVE.
VMM_HW_ARCH_SCEMI	SceMi-compliant platform (not yet supported).

Example 1-2 Selecting Target Platform

```
% vcs +define+VMM_HW_ARCH_NULL ...
```

When using OpenVera, it is important that the target architecture symbol be defined for both the testbench in OpenVera and the hardware-assisted portion of the testbench.

Example 1-3 Selecting Target Platform with OpenVera

```
% vcs -ntb +define+VMM_HW_ARCH_ZEBU -ntb_define  
VMM_HW_ARCH_ZEBU ...
```

Hardware-assisted Portion

When compiling the hardware-assisted portion of the design and testbench for the target platform, you must define the VMM_HW_SYNTHESIS_ON symbol in addition to the target platform

selection symbol. When you define this symbol, only synthesizable Verilog-1995 compliant code is supplied by the `vmm_hw_rtl.sv` file.

Example 1-4 Compiling Assisted Portion

```
% <compiler command>
  +define+VMM_HW_ARCH_SCEMI+VMM_HW_SYNTHESIS_ON \
  bfm.v dut.v ...
```

Simulated Portion

The verification environment must first create an instance of a platform-specific access class that corresponds to the target platform. The name of the class is specific to the target platform and is either “[vmm_hw_arch_null](#)” or “[vmm_hw_arch_zebu](#)”.

Instantiating the platform access class may require additional platform-specific statements to properly initialize the communication between the simulator and the hardware assistance platform.

To make sure testbench code is portable across different platforms, it is recommended that the instantiation of the platform-specific access class, and any other required platform-specific statements, be controlled using the symbols defined in [Table 1-1](#). The instantiated class should then be assigned to a variable of type “[vmm_hw_arch](#)”.

Example 1-5 Selecting Target Platform Access Class

```
vmm_hw_arch platform;

`ifdef VMM_HW_ARCH_NULL
  vmm_hw_arch_null arch = new();
`endif

`ifdef VMM_HW_ARCH_ZEBU
  vmm_hw_arch_zebu arch;
```

```

    ZEBU_board brd = new();

    brd.open(...);
    brd.init(...);
    arch = new(brd);
`endif

platform = arch;

```

When compiling the simulated portion of the testbench for the target platform, the `VMM_HW_SYNTHESIS_ON` symbol must **NEVER** be defined. When undefined, non-synthesizable SystemVerilog code required for the proper operation of the library is supplied by the `vmm_hw.sv` file.

It is also necessary to compile the entire testbench, including the synthesizable section. It is not necessary to include the DUT.

Example 1-6 Compiling Simulated Portion in SV

```

% vcs -sverilog -ntb_opts rvm \
    +define+VMM_HW_ARCH_SCEMI \
    bfm.v tb_top.v tb_env.sv ...

```

Example 1-7 Compiling Simulated Portion in OV

```

% vcs -ntb -sverilog -ntb_opts rvm \
    -ntb_define VMM_HW_ARCH_SCEMI \
    $VCS_HOME/etc/rvm/vmm_hw.vrp \
    bfm.v tb_top.v tb_env.sv ...

```


2

Implementing Hardware-Assisted Testbenches

A hardware-assisted testbench is divided between a synthesized lower layer and simulated upper layers. The synthesized layer executes transactions under the command of the simulated upper layers. It also reports observed transaction and status back to the upper layers.

The transaction-level interface between the synthesized and simulated portions of the testbench is made up of fixed-width unidirectional *input* and *output* ports. An input port sends messages carrying transaction-level information from the simulated testbench layers to the hardware-assisted layer. An output port sends messages from the hardware-assisted layer to the simulated layers. The process for how transaction-level information is mapped to one or more messages is user-defined and not specific to the VMM

hardware abstraction layer or the hardware platform. The process for how messages are relayed between the simulator and the hardware platform is platform-specific and not relevant.

There can be any number of input and output ports. However, the hardware platform may place some restrictions on their number, width or total width.

Messages

This section presents guidelines for implementing efficient and portable transaction-level interfaces between the simulated and hardware-assisted portions of the verification environment.

The frequency of message exchange shall be minimized.

The simulator is much slower than the hardware platform. You will obtain better runtime performance if the hardware-assisted portion of the testbench and design are able to execute with little or no intervention from the simulated portion of the testbench. Therefore, messages between the simulated testbench and the synthesized testbench should be at the highest applicable level of abstraction.

The synthesized layer of the testbench interprets messages to execute transactions. The format of messages is arbitrary and is defined by the behavior of the synthesized layer. Therefore, the format should be defined to minimize the number of messages needed.

For example, a synthesized bus-functional model could simply apply whatever value it receives as a message to a set of output pins, then send the sampled values of input pins back to the testbench. This

would be a mechanism similar to a hardware pin-level tester. However, this approach, although very flexible, requires that two messages be sent at every clock cycle.

A message may represent an entire transaction.

If a transaction can be described within the maximum allowable message size, a message can be a complete transaction.

For example, a small transaction descriptor could be translated into a message by packing its content into the message, as shown in [Example 2-1](#). The synthesized transactor would then interpret the concatenated bits in the message to identify the transaction to be executed.

Example 2-1 Packing a Small Transaction Descriptor Onto a Message

```
in_chan.activate(tr);
tr.byte_pack(bytes);
inp.send({bytes[0], bytes[1], bytes[2], bytes[3],
          bytes[4], bytes[5], bytes[6], bytes[7]});
outp.receive(msg);
tr.data = msg[32:1];
tr.is_ok = msg[0];
in_chan.complete();
```

A transaction may be represented as a sequence of messages.

It may not be possible to fully describe a transaction within a single message. A transaction can be represented using a fixed or variable number of consecutive messages.

As shown in [Example 2-2](#), an ethernet frame, although transmitted 4 bits at a time, could be transferred to a synthesized bus-functional model 4 bits at a time, with a byte-valid indication. A message with at least one invalid byte would be interpreted as the last message in the transaction.

Example 2-2 Variable Length Transaction Messages

```
in_chan.activate(fr);
fr.byte_pack(bytes);
bval = 4'b0000;
foreach (bytes[i]) begin
    int n = i % 4;
    msg[n*8:+8] = bytes[i];
    bval[n] = 1'b1;
    if (n == 3) begin
        inp.send({bval, msg});
        bval = 4'b0000;
    end
end
inp.send({bval, msg});
in_chan.complete();
```

Message ordering can only be guaranteed on a single port.

A transactor may have more than one input or output port. A synthesizable testbench will usually have several. The hardware assistance platform is responsible for transporting messages between the hardware platform and the simulator. The order in which messages on concurrent message ports are transported is not guaranteed. For example, messages generated several clock cycles apart in the hardware platform may be transported at the same time to the simulator.

If message ordering is important, they should be transmitted in a single port or the testbench should include higher-level functionality to ensure that the proper ordering is provided.

Messages may not be received as soon as they are sent.

There is no implicit acknowledgment of messages being received between the simulated and the hardware-assisted portions of the testbench. The hardware assistance platform will likely be able to buffer several messages from the simulator before physically transmitting them to the hardware platform. If explicit confirmation that a transaction has been completed is required, an explicit acknowledgement message must be sent from the hardware platform back to the simulator.

A transaction may be a command to internally generate or accumulate transactions.

For many tests, it is not necessary to control the exact content of a DUT transaction. A synthesized bus-functional model may be able to generate or check well-formed transactions on its own. The messages exchanged between the simulated layers and synthesized layers could simply be to start and stop internal stimulus generation, or to report that an error has been observed.

Different generation strategies can be used:

- Pseudo-random using an LFSR
- Recurring pattern using an FSM
- Data extracted from a local memory

Similarly, different checking strategies can be used:

- Computing then checking a checksum
- Writing data into a local memory

There shall be a "reset-req" input message.

When using a hardware-assistance platform, the hardware-assisted portion will have been reset and be ready to run long before the simulator will have had a chance to start executing the simulated portions of the testbench. Therefore, it is odd that such a message is required.

Remember, the goal is to create a verification environment that will be portable, as-is, to different hardware-assistance platforms, including one that is purely simulated. In the latter case, the design and synthesized testbench layer will not be automatically reset and must be explicitly reset by the testbench.

It must also be possible to verify that the design recovers from a hardware reset during normal operations. To verify the correctness of this operation, it must be possible to apply a hardware reset at anytime. Therefore, a "reset-req" message is necessary.

The simulated testbench shall wait until a "reset-done" message has been received before proceeding.

When using a hardware platform, the DUT will have completed its reset sequence by the time the simulated testbench has had the time to react to the completion of the "reset-req" message. However, for the testbench to remain compatible with a purely simulated platform (e.g., using the "null" architecture or EVE's zTide), there will be a real delay between the completion of the "reset-req" message and the actual completion of the DUT reset.

To ensure that the test proceeds only when the DUT is ready, regardless of the duration of the reset sequence, the simulated testbench must wait until the reset operation has been acknowledged before proceeding.

The simulated testbench shall not directly access anything in the synthesized portion of the testbench nor the DUT.

Although this is possible when using the "null" architecture, this is not a capability that is available when using a hardware platform.

A direct access to the DUT will be reported as an error at compile time because the DUT is not included when compiling the simulated portion.

A direct access to the synthesized portion of the testbench will **not** be reported as an error at compile time because that portion of the testbench is also compiled when compiling the simulated portion. Furthermore, the functionality of the synthesized portion is disabled in the simulated portion, so any direct access will not operate properly.

There should be an environment support transactor.

Every DUT will need some low-level signal management. These low-level support tasks include applying hardware reset, setting external mode pins, etc... A transactor should be created to provide these low-level services. This low-level support transactor could implement the reset-request and reset-ack messages described earlier.

Hardware-Assisted Layer

This section specifies guidelines for implementing the synthesized layer and DUT.

Top-level Module

This section provides guidelines for implementing the top-level module that encapsulates the entire synthesized—and thus hardware-assisted—portion of the testbench and the DUT.

The top-level module shall contain only instantiations

The top-level module is used to specify the connectivity between the DUT, clock generators and synthesizable transactors. There must not be any synthesizable RTL code in the top-level module. All RTL code must be encapsulated either as part of a [“Synthesizable Transactors”](#) or the DUT.

Instances of the `vmm_hw_clock` module shall be located in the top-level module

This is a corollary of the previous guideline. Clock generators must not be located inside transactors nor the DUT (see [“`vmm_hw_clock`” on page A-9](#)).

The DUT instance shall be included only if the `VMM_HW_INCL_DUT` symbol is defined

This is to prevent a syntax error caused by a missing module instantiation when compiling the simulated portion of the testbench.

Example 2-3 DUT Instance

```
module tb_top;
    ...
    `ifdef VMM_HW_INCL_DUT
        design dut(...);
    `endif
endmodule
```

Synthesizable Transactors

This section provides guidelines for implementing synthesizable transactors suitable for hardware-assistance platforms. Of course, it is an implicit requirement that the transactors be implemented using properly synthesizable RTL code.

Transactors shall be encapsulated in a module.

It must be possible to synthesize a transactor on its own. This implies that there must not be any transactor code inlined in the top-level module.

Transactor module pins shall be limited to the physical protocol-related signals.

The only external connectivity a synthesizable transactor must present is to the DUT. All communications to the upper layers of the testbenches must be encapsulated within the transactor.

Instances of the `vmm_hw_in_if` and `vmm_hw_out_if` module must be located inside transactors.

The pins of a transactor are limited to the physical signals involved in the protocol implemented by that transactor. All communication with the upper layers of the testbench must be encapsulated in the transactor module. Therefore, all input and output message interfaces must be instantiated inside transactor modules (see [“vmm_hw_in_if” on page A-3](#) and [“vmm_hw_out_if” on page A-6](#)).

Example 2-4 Message Interface Instances

```
module hw_reset_bfm(rst);  
    output rst;  
    ...  
endmodule
```

```

    vmm_hw_in_if #(1) rst_req_if(...);
    vmm_hw_out_if #(1) rst_ack_if(...);
    ...
endmodule

```

Transactors shall be sensitive to the posedge of uclk.

The "uclk" signal, provided by vmm_hw_in_if and vmm_hw_out_if instances, is an uncontrolled clock signal that never stops (see [“vmm_hw_in_if” on page A-3](#) and [“vmm_hw_out_if” on page A-6](#)). The controlled clock signals, provided by vmm_hw_clock instances, are reserved for the DUT (see [“vmm_hw_clock” on page A-9](#)). Because all message and clock control signals are valid on the posedge of the uncontrolled clock, the BFM's should be similarly sensitive.

If a BFM must be synchronized with a DUT, clock domain can be implemented using the uncontrolled clock and clock control signals as shown in [Example 2-5](#).

Example 2-5 Functionality Sensitive to posedge and negedge of DUT Clock

```

always @(posedge uclk)
begin
    if (cclk_en) begin
        ...
    end
end

always @(posedge uclk)
begin
    if (cclk_neg_en) begin
        ...
    end
end

```

Transactors should disable the DUT clock instead of stalling the protocol.

If messages are not exchanged fast enough between the simulator and the hardware platform, it will be necessary to throttle the physical interface between the DUT and the synthesized transactors to let the simulator catch up.

It is better to disable the clock to the DUT and hold it in its present state than to stall the protocol by creating some "invalid" or "hold" condition. This will allow the same cycle-by-cycle behavior to be reproduced at different clock speeds and in a purely simulated environment. Stalling the protocol will introduce an unpredictable number of idle states which may be very difficult to reproduce in a different environment. These idle states may also prevent the DUT from reaching interesting corner cases because its interfaces are not stressed as fast as possible.

Transactors should have a "protocol stall" message.

This guideline is a corollary of the previous guideline. Stalling the protocol should be treated as just another transaction that the transactor can perform. This way, the DUT will be exercised in a reproducible and predictable way.

Instances of the `vmm_hw_clock_control` module must be located inside transactors.

The pins of a transactor are limited to the physical signals involved in the protocol implemented by that transactor. Clock control signals are not part of the protocol and are handled implicitly by the hardware platform. Therefore, all clock control interfaces must be instantiated inside transactor modules (see [“vmm_hw_clock_control” on page A-11](#)).

Transactors shall not assume that controlled clocks are enabled right away.

The hardware platform does not guarantee that controlled clocks will be enabled in the next uclk cycle after asserting the "rdy_for_cclk" signal. This may be due to inherent delay in the hardware platform itself or other BFMs preventing the controlled clock from being enabled. The "cclk_en" signal must be used to confirmed that the controlled clock has been enabled.

Simulated Portion

The simulated portion of a testbench communicates with the hardware-assisted portion through a class instance. The class instance, called an *input* or *output message port*, provides a constant-width communication channel to a corresponding *input* or *output message interface* in the hardware-assisted portion.

Input and output message ports cannot be instantiated directly. The verification environment must first create an instance of a platform-specific access class that corresponds to the target platform as shown in [Example 1-5](#). The `vmm_hw_arch::create_in_port()` and `vmm_hw_arch::create_out_port()` methods in the platform access class are used to create instances of input or output message ports, as shown in [Example 2-6](#) (see "[vmm_hw_arch::create_in_port\(\)](#)" on [page A-24](#) and "[vmm_hw_arch::create_out_port\(\)](#)" on [page A-26](#)).

As the last step of the `vmm_env::build()` step, the environment must call the `vmm_hw_arch::init_sim()` method of the platform-specific access class (see "[vmm_hw_arch::init_sim\(\)](#)" on [page A-28](#)).

Example 2-6 Reset Message Ports

```
class hw_env extends vmm_env;
  vmm_hw_arch      hw_platform;
  vmm_hw_in_port  rst_req_prt;
  vmm_hw_out_port rst_ack_prt;
  ...
function new();
  super.new();
  begin
    `ifdef VMM_HW_ARCH_NULL
      vmm_hw_arch_null arch = new();
    `endif
    `ifdef VMM_HW_ARCH_SCEMI
      vmm_hw_arch_sceci arch = new();
    `endif
    this.hw_platform = arch;
  end
  ...
endfunction: new
...
function void build();
  super.build();
  this.rst_req_prt = this.hw_platform.create_in_port(
    tb_top.hw_rst.rst_req_if.vitf);
  this.rst_ack_prt = this.hw_platform.create_out_port(
    tb_top.hw_rst.rst_ack_if.vitf);
  ...
  this.hw_platform.init_sim();
endfunction: build
...
endclass: hw_env
```


A

Class Library Reference

This appendix details the classes and components available in the VMM hardware abstraction layer application package used to build hardware-assisted verification environments.

The classes are documented in alphabetical order. The methods in each class are documented in a logical order, where methods that accomplish similar results are documented sequentially. A summary of all available methods with cross references to the page where their detailed documentation can be found is provided at the beginning of each class specification.

Developers of hardware-assisted verification platforms should consult, [Appendix B, "Platform Integration"](#), for more details on how to integrate a new platform in this application package.

Component Summary

- [vmm_hw_in_if](#) [page A-3](#)

- [vmm_hw_out_if](#) page A-6
- [vmm_hw_clock](#) page A-9
- [vmm_hw_clock_control](#) page A-11
- [vmm_hw_arch](#) page A-21
- [vmm_hw_arch_null](#) page A-29
- [vmm_hw_arch_zebu](#) page A-31
- [vmm_hw_in_port](#) page A-33
- [vmm_hw_out_port](#) page A-38

vmm_hw_in_if

Synthesizable endpoint in simulator-to-hardware communication channel.

Verilog

```
module vmm_hw_in_if(rx_rdy, tx_rdy, msg, uclk, urst);  
  
    parameter          width = 1;  
  
    input              rx_rdy;  
    output             tx_rdy;  
    output [width-1:0] msg;  
    input              uclk;  
    input              urst
```

Description

An instance of this module creates a platform-side message interface for a simulation-to-hardware-platform communication channel.

Parameters and Pins

width

Number of bits transferred in each message.

rx_rdy

When set, indicates that the hardware-assisted testbench is ready to transfer a message.

tx_rdy

When set, indicates that the value on the *msg* pin is valid and that the simulated testbench is ready to transfer a message.

msg

Message from the simulation to the hardware-assisted platform. A message is transferred at every rising edge of *uclk* when both *rx_rdy* and *tx_rdy* are asserted and *urst* is de-asserted.

uclk

Uncontrolled clock provided by an instance of the “[vmm_hw_clock_control](#)” module.

urst

Uncontrolled reset provided by an instance of the “[vmm_hw_clock_control](#)” module.

Example

Example A-1

```
module my_bfm (output reg push_req_n, pop_req_n, ..., input
rst);
    ...
    wire uclk, urst;
    wire [31:0] drv_msg;
    reg        drv_rx_rdy;
    wire        drv_tx_rdy;

    vmm_hw_clock_control cc(uclk, urst, ...);
    vmm_hw_in_if #(32) msg1(drv_rx_rdy, drv_tx_rdy, drv_msg,
uclk, urst);
    ...
endmodule

class my_env extends vmm_env ;
    ...
endclass
```

```
vmm_hw_arch arch;  
...  
function void build() ;  
    super.build();  
    vmm_hw_in_port ip;  
    ...  
    ip = arch.create_in_port(tb_top.bfm_write.msg1.vitf, "");  
    ...  
endfunction  
endclass
```

vmm_hw_out_if

Synthesizable endpoint in hardware-to-simulator communication channel.

Verilog

```
module vmm_hw_out_if(tx_rdy, rx_rdy, msg, uclk, urst);  
  
    parameter          width = 1;  
    parameter          pri   = 10;  
  
    input              tx_rdy;  
    output             rx_rdy;  
    input  [width-1:0] msg;  
    input              uclk;  
    input              urst
```

Description

An instance of this module creates a platform-side message interface for a hardware-platform-to-simulator communication channel.

Parameters and Pins

width

Number of bits transferred in each message.

pri

Message priority.

tx_rdy

When set, indicates that the value on the *msg* pin is valid and that the hardware-assisted testbench is ready to transfer a message.

rx_rdy

When set, indicates that the simulated testbench is ready to transfer a message.

msg

Message from the hardware-assisted platform to the simulation. A message is transferred at every rising edge of *uclk* when both *rx_rdy* and *tx_rdy* are asserted and *urst* is de-asserted.

uclk

Uncontrolled clock provided by an instance of the “[vmm_hw_clock_control](#)” module.

urst

Uncontrolled reset provided by an instance of the “[vmm_hw_clock_control](#)” module.

Example

Example A-2

```
module my_bfm (output reg push_req_n, pop_req_n, ..., input
rst);
    ...
    reg [31:0] mon_msg;
    reg mon_tx_rdy;
    wire mon_rx_rdy;

    vmm_hw_clock_control cc(uclk, urst, ...);
    vmm_hw_in_if #(32) msg1(drv_rx_rdy, drv_tx_rdy, drv_msg,
uclk, urst);
    ...
endmodule
```

```

    vmm_hw_out_if #(32) msg2(mon_tx_rdy, mon_rx_rdy, mon_msg,
uclk, urst);
    ...
endmodule

class my_env extends vmm_env ;
    ...
    vmm_hw_arch arch;
    ...
function void build() ;
    super.build();
    vmm_hw_out_port op;
    ...
    op = arch.create_out_port(tb_top.bfm_write.msg2.vitf,
"");
    ...
endfunction
endclass

```

vmm_hw_clock

Synthesizable clock source

Verilog

```
module vmm_hw_clock(cclk, crst);  
    parameter clock_num = 1;  
  
    output cclk;  
    output crst;
```

Description

An instance of this module creates a controlled clock source.

Parameters and Pins

clock_num

Unique positive integer identifier for the clock domain.

cclk

Controlled clock that can be used to drive a clock input on the DUT.

crst

Controlled reset that can be used to drive a reset input on the DUT.

The synthesizable portion of the testbench is loaded—but not exercised—by the simulator. This "cclk" and "crst" signals are therefore not driven and remain at high-impedance in the simulator when an actual hardware platform (i.e., not the *null* platform) is used. These signals are always properly driven in the hardware platform.

Example

Example A-3

```
module tb_top;
    ...
    wire cclk, crst, crstn;
    ...
    my_bfm bfm_write(push_req_n,pop_req_n,...,rst);
    vmm_hw_clock tb_clk(cclk, crst, crstn);
    hw_reset hw_rst(rst);

    `ifdef VMM_HW_INCL_DUT
        my_dut dut(.clk(cclk),
                  .rst(rst),
                  ...
                  .data_out(data_out));
    `endif

endmodule

...
module my_dut (clk, rst,...,data_out);
    ...
endmodule

...
module hw_reset(output rst);
    ...
    vmm_hw_clock_control clk1(...);
    ...
endmodule
```

vmm_hw_clock_control

Synthesizable clock controller

Verilog

```
module vmm_hw_clock_control(uclk, urst,
                           rdy_for_cclk, cclk_en,
                           rdy_for_cclk_neg, cclk_neg_en);
    parameter clock_num = 1;

    output uclk;
    output urst;
    input  rdy_for_cclk;
    output cclk_en;
    input  rdy_for_cclk_neg;
    output cclk_neg_en;
```

Description

An instance of this module creates a controlled clock domain.

Parameters and Pins

clock_num

Unique positive integer identifier for the clock domain.

uclk

Uncontrolled clock used by synthesizable transactors.

urst

Uncontrolled reset used by synthesizable transactors.

`rdy_for_cclk`

When de-asserted, prevents a rising edge on the corresponding controlled clock from occurring.

`cclk_en`

When asserted at the rising edge of *uclk*, indicates that a rising edge on the corresponding controlled clock is about to occur.

`rdy_for_cclk_neg`

When de-asserted, prevents a falling edge on the corresponding controlled clock from occurring.

`cclk_en_neg`

When asserted at the rising edge of *uclk*, indicates that a falling edge on the corresponding controlled clock is about to occur.

The synthesizable portion of the testbench is loaded—but not exercised—by the simulator. This "uclk" and "urst" signals are therefore not driven and remain at high-impedance in the simulator when an actual hardware platform (i.e., not the *null* platform) is used. These signals are always properly driven in the hardware platform.

Example

Example A-4

```
module my_bfm (output reg push_req_n, pop_req_n, ..., input
rst);
    wire uclk, urst, posedge_en, negedge_en;
    wire posedge_rdy = 1;
    wire negedge_rdy = 1;
    ...
    vmm_hw_clock_control clk1(uclk, urst, posedge_rdy,
posedge_en,
```

```
...
negedge_rdy, negedge_en);
endmodule
```

OpenVera Interface Bindings

Macros for creating bindings between the OpenVera testbench and instances of the hardware-assisted verification components.

These macros must be instantiated at the file level, outside of any class or program.

Applicable to OpenVera only.

Binding Macros

- [vmm_hw_in_if_bind](#) page A-15
- [vmm_hw_out_if_bind](#) page A-17
- [vmm_hw_clock_bind](#) page A-19
- [vmm_hw_clock_control_bind](#) page A-20

vmm_hw_in_if_bind

Create a binding to a “vmm_hw_in_if” instance.

SystemVerilog

N/A

OpenVera

```
vmm_hw_in_if_bind(identifier    name,  
                  vmm_hw_in_if path  
                  integer      size)
```

Description

Creates a virtual port binding to an instance of a platform-side input message interface.

IMPORTANT: because this is a macro, it is important that it not be terminated with a semi-colon, otherwise a syntax error will be created.

name

A globally unique valid identifier.

path

Absolute hierarchical reference to a “vmm_hw_in_if” module instance.

size

The number of bits in the `vmm_hw_in_if` module instance.

vmm_hw_out_if_bind

Create a binding to a “vmm_hw_out_if” instance.

SystemVerilog

N/A

OpenVera

```
vmm_hw_out_if_bind(identifier    name,  
                  vmm_hw_out_if path  
                  integer      size)
```

Description

Creates a virtual port binding to an instance of a platform-side output message interface.

IMPORTANT: because this is a macro, it is important that it not be terminated with a semi-colon, otherwise a syntax error will be created.

name

A globally unique valid identifier.

path

Absolute hierarchical reference to a “vmm_hw_out_if” module instance.

size

The number of bits in the “[vmm_hw_out_if](#)” module instance.

vmm_hw_clock_bind

Create a binding to a “vmm_hw_clock” instance.

SystemVerilog

N/A

OpenVera

```
vmm_hw_clock_bind(name,  
                  vmm_hw_clock path)
```

Description

Creates a virtual port binding to an instance of a synthesizable clock source.

IMPORTANT: because this is a macro, it is important that it not be terminated with a semi-colon, otherwise a syntax error will be created.

name

Name of the virtual port binding.

path

Absolute hierarchical reference to a “vmm_hw_clock” module instance.

vmm_hw_clock_control_bind

Create a binding to a “[vmm_hw_clock_control](#)” instance.

SystemVerilog

N/A

OpenVera

```
vmm_hw_clock_control_bind(name,  
                           vmm\_hw\_clock\_control path)
```

Description

Creates a virtual port binding to an instance of a synthesizable clock controller.

IMPORTANT: because this is a macro, it is important that it not be terminated with a semi-colon, otherwise a syntax error will be created.

name

Name of the virtual port binding.

path

Absolute hierarchical reference to a “[vmm_hw_clock_control](#)” module instance.

vmm_hw_arch

Base class for hardware-assistance platform access class. Cannot be used directly. One of the platform-specific extensions must be used instead.

Platform Access Classes

- [vmm_hw_arch_null](#) page A-29
- [vmm_hw_arch_zebu](#) page A-31
- [vmm_hw_arch::clk_control\(\)](#) page A-22
- [vmm_hw_arch::create_in_port\(\)](#) page A-24
- [vmm_hw_arch::create_out_port\(\)](#) page A-26
- [vmm_hw_arch::init_sim\(\)](#) page A-28

vmm_hw_arch::clk_control()

Associate a clock controller with a clock source.

SystemVerilog

```
virtual function void clk_control(  
    virtual vmm_hw_clock          clk,  
    virtual vmm_hw_clock_control ctl);
```

OpenVera

```
virtual task clk_control(  
    vmm_hw_clock_bind          clk,  
    vmm_hw_clock_control_bind ctl);
```

Description

Bind an instance of a synthesizable clock controller with an instance of a synthesizable controlled clock source. A single clock source may be controlled by more than one clock controller. A clock controller instance can only control one clock source.

clk

In SV, an absolute hierarchical reference to the “vmm_hw_clock” module instance controlled by the specified clock control module. In OV, the name of a “vmm_hw_clock_bind”, bound to a “vmm_hw_clock” module instance.

ctl

In SV, an absolute hierarchical reference to a “[vmm_hw_clock_control](#)” module instance that controls the specified clock source. In OV, the name of a “[vmm_hw_clock_control_bind](#)”, bound to a “[vmm_hw_clock_control](#)” module instance.

Example

Example A-5

```
class my_env extends vmm_env;
    ...
    vmm_hw_arch_null arch_null;
    vmm_hw_arch arch;
    ...
    `ifdef VMM_HW_ARCH_NULL
        begin
            vmm_hw_arch_null a = new;
            this.arch = a;
        end
    `endif
    ...
    function void build() ;
        ...

    arch.clk_control(tb_top.tb_clk.vitf, tb_top.hw_rst.bfm.tb_c
lk.vitf);
        ...
    endfunction
    ...
endclass
```

vmm_hw_arch::create_in_port()

Create and associate a simulation-side message port with a platform-side message interface.

SystemVerilog

```
virtual function vmm_hw_in_port create_in_port(  
    virtual vmm_hw_in_if itf,  
    string name = "");
```

OpenVera

```
virtual function vmm_hw_in_port create_in_port(  
    vmm_hw_in_if_bind itf,  
    string name);
```

Description

Create a simulation-side message port for a simulation-to-platform communication channel. The platform-side message interface is specified as the absolute hierarchical name to the corresponding instance of the “vmm_hw_in_if” module instance.

itf

In SV, an absolute hierarchical reference to the “vmm_hw_in_if” module instance corresponding to the platform-side endpoint. In OV, the identifier of a “vmm_hw_in_if_bind”, bound to a “vmm_hw_in_if” module instance.

name

User-specified interface name that may be required by some architectures. See the documentation for the platform-specific extension of the “[vmm_hw_arch](#)” class for the need and requirements of this argument. Optional in SV only.

Example

Example A-6

```
class my_env extends vmm_env;
  ...
  vmm_hw_in_port ip_port;
  ...
  function void build() ;
    ...
    this.ip_port =
arch.create_in_port(tb_top.bfm.req_in_if.vitf,"ip_port");
    ...
    // req_in_if interface at HW platform
  endfunction
  ...
endclass
```

vmm_hw_arch::create_out_port()

Create and associate a simulation-side message port with a platform-side message interface.

SystemVerilog

```
virtual function vmm_hw_out_port create_in_port(  
    virtual vmm_hw_out_if itf,  
    string name = "");
```

OpenVera

```
virtual function vmm_hw_out_port create_in_port(  
    vmm_hw_out_if_bind itf  
    string name);
```

Description

Create a simulation-side message port for a platform-to-simulation communication channel. The platform-side message interface is specified as the absolute hierarchical name to the corresponding instance of the “vmm_hw_out_if” module instance.

itf

In SV, an absolute hierarchical reference to the “vmm_hw_out_if” module instance corresponding to the platform-side endpoint. In OV, the identifier of a “vmm_hw_out_if_bind”, bound to a “vmm_hw_out_if” module instance.

name

User-specified name that may be required by some architectures. See the documentation for the platform-specific extension of the “[vmm_hw_arch](#)” class for the need and requirements of this argument. Optional in SV only.

Example

Example A-7

```
class my_env extends vmm_env ;
    ...
    vmm_hw_out_port out_port;
    ...
    function void build();
        ...
        this.out_port = arch.create_out_port
(tb_top.bfm.req_out_if.vitf,"out_port");
        ...
        // req_out_if interface at HW platform
    endfunction
    ...
endclass
```

vmm_hw_arch::init_sim()

Initialize the platform to get ready for simulation.

SystemVerilog

```
virtual function void init_sim();
```

OpenVera

```
virtual task init_sim();
```

Description

Perform any post-connection task required by the hardware assistance platform to make it ready for simulation.

Example

Example A-8

```
class my_env extends vmm_env ;  
    ...  
    function void build() ;  
        ...  
        arch.init_sim();  
        ...  
    endfunction  
    ...  
endclass
```

vmm_hw_arch_null

Simulation-only pseudo-platform access class.

SystemVerilog

```
vmm_hw_arch_null arch = new;
```

OpenVera

```
vmm_hw_arch_null arch = new;
```

Description

Create a platform access class for a simulation-only platform. This platform is used to develop testbenches and testcases and debug the DUT without a hardware-assistance platform.

Requires that the `VMM_HW_ARCH_NULL` compile-time symbol be defined.

The connection between a port instance and an interface instance is done using a virtual interface.

The optional user-specified interface name is “`vmm_hw_arch::create_in_port()`” and “`vmm_hw_arch::create_out_port()`” is not used.

Example

Example A-9

```
class my_env extends vmm_env ;
```

```
...
vmm_hw_arch arch;
...
function new();
...
`ifdef VMM_HW_ARCH_NULL
    vmm_hw_arch_null arch_null = new;
    this.arch = arch_null;
`endif
...
endfunction
...
endclass
```

vmm_hw_arch_zebu

Platform access class for the Zebu platform from EVE.

SystemVerilog

```
ZEBU_board          brd = new(...);  
vmm_hw_arch_zebu arch = new(brd);
```

OpenVera

```
ZEBU_board          brd = new(...);  
vmm_hw_arch_zebu arch = new(brd);
```

Description

Create a platform access class for a Zebu or zTide platform from EVE. Requires that the `VMM_HW_ARCH_ZEBU` compile-time symbol be defined.

The name used to connect a message port to a message interface is the full HDL path of the interface instance specified in the `“vmm_hw_arch::create_in_port()”` and `“vmm_hw_arch::create_out_port()”` methods, less the top-level module name. The same name should be used in the DVE file.

The optional user-specified interface name in `“vmm_hw_arch::create_in_port()”` and `“vmm_hw_arch::create_out_port()”` is not used.

Example

Example A-10

```
`ifndef VMM_HW_ARCH_ZEBU
  class vmm_hw_arch_zebu extends vmm_hw_arch;
    ...
    virtual function void clk_control(virtual
vmm_hw_clock_itf clk,
                                     virtual
vmm_hw_clock_control_itf ctl);
    //User defined logic must be placed here.
    ...
  endfunction
  ...
endclass
`endif

class my_env extends vmm_env;
  ...
  vmm_hw_arch_zebu arch_zebu;
  ...
  function void build() ;
    ...
    `ifndef VMM_HW_ARCH_ZEBU
      arch_zebu.clk_control

(tb_top.tb_clk.vitf,tb_top.hw_rst.bfm.tb_clk.vitf);
    `endif
    ...
  endfunction
  ...
endclass
```

vmm_hw_in_port

Simulation-side message port class for a simulation-to-hardware-platform communication channel.

An instance of this class is created using the “[vmm_hw_arch::create_in_port\(\)](#)” method and is associated with an instance of a “[vmm_hw_in_if](#)” module.

Summary

- [vmm_hw_in_port::is_rdy\(\)](#) page A-34
- [vmm_hw_in_port::wait_is_rdy\(\)](#) page A-35
- [vmm_hw_in_port::send\(\)](#) page A-37

vmm_hw_in_port::is_rdy()

Check if the platform-side message interface is ready to receive a message.

SystemVerilog

```
function bit is_rdy();
```

OpenVera

```
function bit is_rdy();
```

Description

Returns TRUE if the *rx_rdy* pin of the “vmm_hw_in_if” module instance corresponding to the message interface associated with this message port is asserted.

Example

Example A-11

```
...
if(ip_port.is_rdy() == 1)
  `vmm_note(log, "Transmitter Port is ready to send data.");
else
  ip_port.wait_is_rdy();
...
```

vmm_hw_in_port::wait_is_rdy()

Wait until the platform-side message interface is ready to receive a message.

SystemVerilog

```
task wait_is_rdy();
```

OpenVera

```
task wait_is_rdy_t();
```

Description

Wait until the *rx_rdy* pin of the “vmm_hw_in_if” module instance corresponding to the message interface associated with this message port is asserted.

Example

Example A-12

```
...
ip_port.wait_is_rdy();
class my_env extends vmm_env ;
    ...
    function void build() ;
        vmm_hw_in_port ip;
        ip = arch.create_in_port(tb_top.bfm_write.msg1.vitf,
    "");
        mst = new ("trans master", 1, ip, gen2mas);
        ...
endclass: my_env
```

```
class my_master extends vmm_xactor;

    local vmm_hw_in_port ip_port;
    function new(string inst,
                 integer stream_id = -1,
                 vmm_hw_in_port port,
                 trans_channel in_chan = null);
    // Call the super task to initialize the xactor
    super.new("MASTER", inst, stream_id) ;
    ip_port = port;
    ip_port.send(32'hAAAAB0B0);
    ...
endclass: my_master
```

vmm_hw_in_port::send()

Send a message to the platform-side message interface.

SystemVerilog

```
task send(bit [ `VMM_HW_MAX_MSG_WIDTH-1:0 ] data);
```

OpenVera

```
task send_t(bit [VMM_HW_MAX_MSG_WIDTH-1:0] data);
```

Description

Send the specified message to the platform-side message interface associated with this message port. If the message interface is not ready to receive the message, this method will wait until the message is transferred.

The default maximum message width is 1024 bits.

vmm_hw_out_port

Simulation-side message port class for a hardware-platform-to-simulation communication channel.

An instance of this class is created using the [“vmm_hw_arch::create_out_port\(\)”](#) method and is associated with an instance of a [“vmm_hw_out_if”](#) module.

Summary

- [vmm_hw_out_port::is_rdy\(\)](#) [page A-39](#)
- [vmm_hw_out_port::wait_is_rdy\(\)](#) [page A-40](#)
- [vmm_hw_out_port::receive\(\)](#) [page A-41](#)

vmm_hw_out_port::is_rdy()

Check if the platform-side message interface is ready to send a message.

SystemVerilog

```
function bit is_rdy();
```

OpenVera

```
function bit is_rdy();
```

Description

Returns TRUE if the *tx_rdy* pin of the “vmm_hw_out_if” module instance corresponding to the message interface associated with this message port is asserted.

Example

Example A-13

```
...
req_prt.send(32'hAAAAB0B0);
...
if(out_port.is_rdy() == 1)
  `vmm_note(log,"Receiver port is ready to receive data.");
else
  out_port.wait_is_rdy();
...
```

vmm_hw_out_port::wait_is_rdy()

Wait until the platform-side message interface is ready to send a message.

SystemVerilog

```
task wait_is_rdy();
```

OpenVera

```
task wait_is_rdy_t();
```

Description

Wait until the *tx_rdy* pin of the “*vmm_hw_out_if*” module instance corresponding to the message interface associated with this message port is asserted.

Example

Example A-14

```
...
if(out_port.is_rdy() == 1)
  `vmm_note(log, "Receiver port is ready to receive data.");
else
  out_port.wait_is_rdy();
...
```

vmm_hw_out_port::receive()

Receive a message from the platform-side message interface.

SystemVerilog

```
task receive(ref bit ['VMM_HW_MAX_MSG_WIDTH-1:0] data
             ref time                                     stamp);
```

OpenVera

```
function bit [63:0]
    receive_t(var bit [VMM_HW_MAX_MSG_WIDTH-1:0] msg);
```

Description

Receive the next message from the platform-side message interface associated with this message port. If the message interface is not ready to send a message, this method will wait until a message is transferred.

The default maximum message width is 1024 bits.

stamp (SV) or return value (OV)

The uncontrolled clock cycle count when the message was transferred through the “[vmm_hw_out_if](#)” module instance.

Example

Example A-15

```
...
bit [1023:0] msg;
```

```

time stamp;
...
ack_prt.wait_is_rdy();

class my_env extends vmm_env ;
...
function void build() ;
vmm_hw_out_port op;
op = arch.create_out_port(tb_top.bfm_write.msg2.vitf,
    "");
mon = new ("trans monitor", 1, op, mon2scb);
...
endclass: my_env

class my_monitor extends vmm_xactor;
local vmm_hw_in_port ip_port;
function new(string inst,
    int stream_id = -1,
    vmm_hw_out_port port,
    trans_channel out_chan = null);
// Call the super task to initialize the xactor
super.new("MONITOR", inst, stream_id) ;
op_port = port;
op_prt.receive(msg, stamp);

if (msg !== 32'hB0B0AAAA) begin
    `vmm_error(log,$psprintf("RX data 'h%0h is not
        matched with TX data.", msg));
end
...
endclass: my_monitor

```

B

Platform Integration

This appendix details how to integrate a new platform in the VMM hardware abstraction layer package and provides documentation for methods not documented in [Appendix A, "Class Library Reference"](#), because they are not used directly by testbench users. It also provides additional information for classes and methods already documented in Appendix A that are involved in supporting and abstracting the hardware assistance platform.

Three new classes must be created to integrate a new hardware assistance platform:

- [vmm_hw_arch_XYZ](#)
- [vmm_hw_in_port_XYZ](#)
- [vmm_hw_out_port_XYZ](#)

All platform-specific code must be guarded by a corresponding `VMM_HW_ARCH_XYZ` compile-time symbol to be defined.

Additional information can be found in [Appendix A, "Class Library Reference"](#).

Component Summary

- [vmm_hw_arch_XYZ](#) page B-3
- [vmm_hw_in_port_XYZ](#) page B-17
- [vmm_hw_out_port_XYZ](#) page B-25

vmm_hw_arch_XYZ

Platform access class for the XYZ platform.

Description

Platform access class, based on [vmm_hw_arch](#), for the XYZ platform.

All platform-specific mechanisms must be encapsulated within this class (for an example, see [vmm_hw_arch_zebu](#)). All virtual methods in this class must be implemented.

Example

Example B-1

```
\ifdef VMM_HW_ARCH_XYZ
    class vmm_hw_arch_XYZ extends vmm_hw_arch;
        function new(...);
        ...
    endclass
\endif
```

Content Summary

- [vmm_hw_arch_XYZ::clk_control\(\)](#) page B-4
- [vmm_hw_arch_XYZ::create_in_port\(\)](#) page B-7
- [vmm_hw_arch_XYZ::create_out_port\(\)](#) page B-10
- [vmm_hw_arch::connect_to\(\)](#) page B-13
- [vmm_hw_arch::init_sim\(\)](#) page B-15

vmm_hw_arch_XYZ::clk_control()

Associate a clock controller with a clock source.

SystemVerilog

```
virtual function void clk_control(  
    virtual vmm_hw_clock_itf clk,  
    virtual vmm_hw_clock_control_itf ctl);
```

OpenVera

Not yet implemented

Description

Bind an instance of a synthesizable clock controller with an instance of a synthesizable controlled clock source. A single clock source may be controlled by more than one clock controller. A clock controller instance can only control one clock source.

clk

Absolute hierarchical reference to the [vmm_hw_clock](#) module instance controlled by the specified clock control module, suffixed with ".vitf".

ctl

Absolute hierarchical reference to a [vmm_hw_clock_control](#) module instance that controls the specified clock source, suffixed with ".vitf".

Implementation

If the platform uses some other means of associating clock controller instances with controlled clock sources, this method may be left empty, display the necessary information to specify this binding, or verify that the binding was specified correctly.

For example, if an external file is used to specify the binding, this method could be used to display the content of that file.

For example, if parameters are used to specify the binding, this method could be used to check that the parameters are assigned consistent and coherent values.

Example

Example B-2

```
`ifndef VMM_HW_ARCH_XYZ
    class vmm_hw_arch_XYZ extends vmm_hw_arch;
        ...
        virtual function void clk_control(virtual
            vmm_hw_clock_itf clk,
            virtual vmm_hw_clock_control_itf ctl);
            //User defined logic must be placed here.
        ...
    endfunction
    ...
endclass
`endif

class my_env extends vmm_env;
    ...
    vmm_hw_arch_xyz arch_xyz;
    ...
    function void build() ;
        ...
        `ifndef VMM_HW_ARCH_XYZ
```

```
        arch_xyz.clk_control(tb_top.tb_clk.vitf,  
        tb_top.hw_rst.bfm.tb_clk.vitf);  
    `endif  
    ...  
endfunction  
...  
endclass
```

vmm_hw_arch_XYZ::create_in_port()

Create and associate a simulation-side message port with a platform-side message interface.

SystemVerilog

```
virtual function vmm_hw_in_port create_in_port(  
    virtual vmm_hw_in_if_itf itf,  
    string name = "");
```

OpenVera

```
virtual function vmm_hw_in_port _create_in_port(  
    vmm_hw_in_if_port itf,  
    string path,  
    integer size,  
    string name = "");
```

Description

Create a simulation-side message port for a simulation-to-platform communication channel. The platform-side message interface is specified as the absolute hierarchical name to the corresponding instance of the `vmm_hw_in_if` module instance.

itf

In SV, absolute hierarchical reference to the `vmm_hw_in_if` module instance corresponding to the platform-side message interface, suffixed with ".vitf". In OV, an open binding that must not be used.

path

(OV only) The path specified in the “[vmm_hw_in_if_bind](#)” macro.

size

(OV only) The size specified in the “[vmm_hw_in_if_bind](#)” macro.

name

Optional user-specified name that can be required by some architectures (see [vmm_hw_arch::connect_to\(\)](#).)

Implementation

This method must create an instance of the [vmm_hw_in_port_XYZ](#) class and return it. Any platform-specific constructs or object must be created within this method or within the [vmm_hw_in_port_XYZ](#) object itself.

If a named-based connection mechanism is used by the platform to associate a simulation-side message port with a platform-side message interface, the [vmm_hw_arch::connect_to\(\)](#) method should be used to determine that name by calling it using:

In SystemVerilog:

```
this.connect_to(itf.path, name);
```

In OpenVera:

```
this.connect_to(path, name);
```

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-3

```
virtual function vmm_hw_in_port create_in_port(  
    virtual vmm_hw_in_if_itf itf,  
    string name = "");  
    vmm_hw_in_port_XYZ inp = new(this.connect_to(itf.name,  
        name));  
    return inp;  
endfunction
```

vmm_hw_arch_XYZ::create_out_port()

Create and associate a simulation-side message port with a platform-side message interface.

SystemVerilog

```
function vmm_hw_out_port create_in_port(  
    virtual vmm_hw_out_if_itf itf,  
    string name = "");
```

OpenVera

```
virtual function vmm_hw_out_port _create_out_port(  
    vmm_hw_out_if_port itf,  
    string path,  
    integer size,  
    string name = "");
```

Description

Create a simulation-side message port for a platform-to-simulation communication channel. The platform-side message interface is specified as the absolute hierarchical name to the corresponding instance of the `vmm_hw_out_if` module instance.

itf

In SV, an absolute hierarchical reference to the `vmm_hw_out_if` module instance corresponding to the platform-side message interface, suffixed with ".vitf". In OV, an open binding that must not be used.

path

(OV only) The path specified in the “[vmm_hw_out_if_bind](#)” macro.

size

(OV only) The size specified in the “[vmm_hw_out_if_bind](#)” macro.

name

Optional user-specified name that can be required by some architectures (see [vmm_hw_arch::connect_to\(\)](#)).

Implementation

This method must create an instance of the [vmm_hw_out_port_XYZ](#) class and return it. Any platform-specific constructs or object must be created within this method or within the [vmm_hw_out_port_XYZ](#) object itself.

If a named-based connection mechanism is used by the platform to associate a simulation-side message port with a platform-side message interface, the [vmm_hw_arch::connect_to\(\)](#) method should be used to determine that name by calling it using:

In SystemVerilog:

```
this.connect_to(itf.path, name);
```

In OpenVera:

```
this.connect_to(path, name);
```

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-4

```
virtual function vmm_hw_out_port create_in_port(  
    virtual vmm_hw_out_if_itf itf,  
    string name = "");  
    vmm_hw_out_port_XYZ inp = new(this.connect_to(itf.name,  
        name));  
    return inp;  
endfunction
```

vmm_hw_arch::connect_to()

Map a HDL path and optional user-specified name to a connection name.

SystemVerilog

```
virtual function string connect_to(string hdl_path,  
                                  string name);
```

OpenVera

```
virtual function string connect_to(string hdl_path,  
                                  string name);
```

Description

Create a name that can be used by platform-specific code to implement a name-based connection of a message port instance (i.e. an instance of a [vmm_hw_in_port](#) or [vmm_hw_out_port](#) class) to a message interface instance (i.e. an instance of a [vmm_hw_in_if](#) or [vmm_hw_out_if](#) module). The name can be derived from the specified HDL path and user-specified name. The implementation is platform-dependent.

By default, the returned name is the full specified HDL path. The user-specified name is not used.

hdl_path

The full hierarchical path of the message interface instance in the HDL model.

name

The optional name specified in the `vmm_hw_arch::create_in_port()` or `vmm_hw_arch::create_out_port()` method.

Example

Example B-5

```
class vmm_hw_arch_XYZ extends vmm_hw_arch;
...
virtual function string connect_to(string hdl_path,string
    name);
    //User defined logic must be placed here.
    return super.connect_to(hdl_path, name); //For Example
...
endfunction
...
endclass
...
`vmm_note(log,$psprintf("Connected XYZ Platform to
    %0s",arch_xyz.connect_to(req_if.path,name)));
...
```

vmm_hw_arch::init_sim()

Initialize the hardware assistance platform.

SystemVerilog

```
virtual function void init_sim();
```

OpenVera

```
virtual task init_sim();
```

Description

Perform any final initialization required by the hardware assistance platform to get it ready for simulation.

If no further initialization is required, this method need not be overloaded.

Example

Example B-6

```
class vmm_hw_arch_XYZ extends vmm_hw_arch;
    ...
    virtual function init_sim();
        //User defined logic must be placed here.
        //Perform any final initialization required by the
        //hardware
        ...
    endfunction
    ...
endclass
...
```

```
this.arch_xyz.init_sim();  
...
```

vmm_hw_in_port_XYZ

Input message port class for the XYZ platform.

Description

Input message port class, based on [vmm_hw_in_port](#), for the XYZ platform.

All platform-specific message-passing mechanism must be encapsulated within this class. All virtual methods in this class must be implemented.

The constructor for this class is not documented and never used directly by users. Instances of this class are created by the [vmm_hw_arch_XYZ::create_in_port\(\)](#) method. Therefore, the constructor interface for this class can be designed according to the arbitrary needs of the platform and the information that need to be passed from the [vmm_hw_arch_XYZ](#) class.

All documented virtual methods must be implemented.

Example

```
class vmm_hw_in_port_XYZ extends vmm_hw_in_port;
...
function new(virtual vmm_hw_in_if_itf itf);
    log.set_instance(itf.path);
    this.itf = itf;
...
endfunction
...
endclass
...
```

```
vmm_hw_in_port_XYZ ip_port;
...
this.ip_port =
arch_xyz.create_in_port(tb_top.bfm.req_in_if.vitf,
    "usr_ip_port");
...
```

Summary

- [vmm_hw_in_port_XYZ::is_rdy\(\)](#) page B-19
- [vmm_hw_in_port_XYZ::wait_is_rdy\(\)](#) page B-21
- [vmm_hw_in_port_XYZ::send\(\)](#) page B-23

vmm_hw_in_port_XYZ::is_rdy()

Check if the platform-side message interface is ready to receive a message.

SystemVerilog

```
virtual function bit is_rdy();
```

OpenVera

```
virtual function bit is_rdy();
```

Description

See [vmm_hw_in_port::is_rdy\(\)](#).

Implementation

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-7

```
class vmm_hw_in_port_XYZ extends vmm_hw_in_port;
...
virtual function bit is_rdy();
    //User defined logic must be placed here.
    return this.itf.rx_rdy;//For Example
...
endfunction
```

```
    ...  
endclass
```

vmm_hw_in_port_XYZ::wait_is_rdy()

Wait until the platform-side message interface is ready to receive a message.

SystemVerilog

```
virtual task wait_is_rdy();
```

OpenVera

```
virtual task wait_is_rdy_t();
```

Description

See [vmm_hw_in_port::wait_is_rdy\(\)](#).

Implementation

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-8

```
class vmm_hw_in_port_XYZ extends vmm_hw_in_port;
...
virtual task wait_is_rdy();
    //User defined logic must be placed here.
    wait(this.itf.rx_rdy == 1'b1); //For Example
...
endtask
```

```
...  
endclass
```

vmm_hw_in_port_XYZ::send()

Send a message to the platform-side message interface.

SystemVerilog

```
virtual task send(bit [`VMM_HW_DATA_WIDTH-1:0] data);
```

OpenVera

```
virtual task send_t(bit [VMM_HW_MAX_MSGDATA_WIDTH-1:0]  
data);
```

Description

See [vmm_hw_in_port::send\(\)](#).

Implementation

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-9

```
class vmm_hw_in_port_XYZ extends vmm_hw_in_port;  
    ...  
    virtual task send(bit [`VMM_HW_DATA_WIDTH - 1:0] data);  
        //User defined logic must be placed here.  
        ...  
        //For Example  
        @(this.itf.ck);
```

```
        this.itf.tx_rdy <= 1'b1;
        ...
        this.itf.msg    <= data;
        ...
    endtask
    ...
endclass
```

vmm_hw_out_port_XYZ

Output message port class for the XYZ platform.

Description

Output message port class, based on [vmm_hw_out_port](#), for the XYZ platform.

All platform-specific message-passing mechanisms must be encapsulated within this class. All virtual methods in this class must be implemented.

The constructor for this class is not documented and never used directly by users. Instances of this class are created by the [vmm_hw_arch_XYZ::create_out_port\(\)](#) method. Therefore, the constructor interface for this class can be designed according to the arbitrary needs of the platform and the information that needs to be passed from the [vmm_hw_arch_XYZ](#) class.

All documented virtual methods must be implemented.

Example

Example B-10

```
class vmm_hw_out_port_XYZ extends vmm_hw_out_port;
...
function new(virtual vmm_hw_out_if_itf itf);
    log.set_instance(itf.path);
    this.itf = itf;
...
endfunction
...
endclass
...
```

```
vmm_hw_out_port_XYZ out_port;  
...  
this.out_port =  
arch_xyz.create_out_port(tb_top.bfm.req_out_if.vitf,  
    "usr_out_port");  
...
```

Summary

- [vmm_hw_out_port_XYZ::is_rdy\(\)](#) page B-27
- [vmm_hw_out_port_XYZ::wait_is_rdy\(\)](#) page B-29
- [vmm_hw_out_port_XYZ::receive\(\)](#) page B-31

vmm_hw_out_port_XYZ::is_rdy()

Check if the platform-side message interface is ready to send a message.

SystemVerilog

```
virtual function bit is_rdy();
```

OpenVera

```
virtual function bit is_rdy();
```

Description

See [vmm_hw_out_port::is_rdy\(\)](#).

Implementation

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-11

```
class vmm_hw_out_port_XYZ extends vmm_hw_out_port;
    ...
    virtual function bit is_rdy();
        //User defined logic must be placed here.
        return this.itf.tx_rdy;//For Example
    ...
endfunction
```

```
    ...  
endclass
```

vmm_hw_out_port_XYZ::wait_is_rdy()

Wait until the platform-side message interface is ready to send a message.

SystemVerilog

```
virtual task wait_is_rdy();
```

OpenVera

```
virtual task wait_is_rdy_t();
```

Description

See [vmm_hw_out_port::wait_is_rdy\(\)](#).

Implementation

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-12

```
class vmm_hw_out_port_XYZ extends vmm_hw_out_port;
...
virtual task wait_is_rdy();
    //User defined logic must be placed here.
    wait(this.itf.tx_rdy == 1'b1); //For Example
...
endtask
```

```
...  
endclass
```

vmm_hw_out_port_XYZ::receive()

Receive a message from the platform-side message interface.

SystemVerilog

```
virtual task receive(ref bit [`VMM_HW_DATA_WIDTH-1:0]
    data ref time stamp);
```

OpenVera

```
virtual function bit [63:0] receive_t(
    var bit [VMM_HW_MAX_MSGDATA_WIDTH-1:0] msg);
```

Description

See [vmm_hw_out_port::receive\(\)](#).

Implementation

The default implementation of this method in the base class must not be called as it simply reports an error about an unimplemented virtual method.

Example

Example B-13

```
class vmm_hw_out_port_XYZ extends vmm_hw_out_port;
    ...
    virtual task receive(ref bit [`VMM_HW_DATA_WIDTH-1:0]
        data, ref time stamp);
        //User defined logic must be placed here.
        //For Example
```

```
        @(this.itf.ck);
        this.itf.rx_rdy <= 1'b1;
        ...
        data = this.itf.msg;
        ...
    endtask
    ...
endclass
```