

# **VMM Hardware Abstraction Layer User Guide**

---

December 2008

VMM HAL Version 2.0 (Synopsys)

# Copyright Notice and Proprietary Information

Copyright © 2008 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 .....	3
Hardware-Assisted Portion .....	3
Simulated Portion .....	4

## 2 Implementing HW-Assisted Testbenches

Messages .....	8
Hardware-Assisted Layer .....	13
Top-Level Module .....	13
Synthesizable Transactors .....	14
Simulated Portion .....	17

## A Class Library Reference

vmm_hw_in_if .....	23
vmm_hw_out_if .....	25
vmm_hw_clock .....	27
vmm_hw_clock_control .....	29
vmm_hw_arch .....	31
vmm_hw_arch::clk_control() .....	32
vmm_hw_arch::create_in_port() .....	33
vmm_hw_arch::create_out_port() .....	35
vmm_hw_arch::init_sim() .....	37

vmm_hw_arch_null .....	38
vmm_hw_in_port .....	39
vmm_hw_in_port::is_rdy() .....	40
vmm_hw_in_port::wait_is_rdy() .....	41
vmm_hw_in_port::send() .....	42
vmm_hw_out_port .....	43
vmm_hw_out_port::is_rdy() .....	44
vmm_hw_out_port::wait_is_rdy() .....	45
vmm_hw_out_port::receive() .....	46

## **B Platform Integration**

vmm_hw_arch_XYZ .....	49
vmm_hw_arch_XYZ::clk_control() .....	50
vmm_hw_arch_XYZ::create_in_port() .....	52
vmm_hw_arch_XYZ::create_out_port() .....	54
vmm_hw_arch::connect_to() .....	56
vmm_hw_arch::init_sim() .....	58
vmm_hw_in_port_XYZ .....	59
vmm_hw_in_port_XYZ::is_rdy() .....	60
vmm_hw_in_port_XYZ::wait_is_rdy() .....	61
vmm_hw_in_port_XYZ::send() .....	62
vmm_hw_out_port_XYZ .....	63
vmm_hw_out_port_XYZ::is_rdy() .....	64
vmm_hw_out_port_XYZ::wait_is_rdy() .....	65
vmm_hw_out_port_XYZ::receive() .....	66

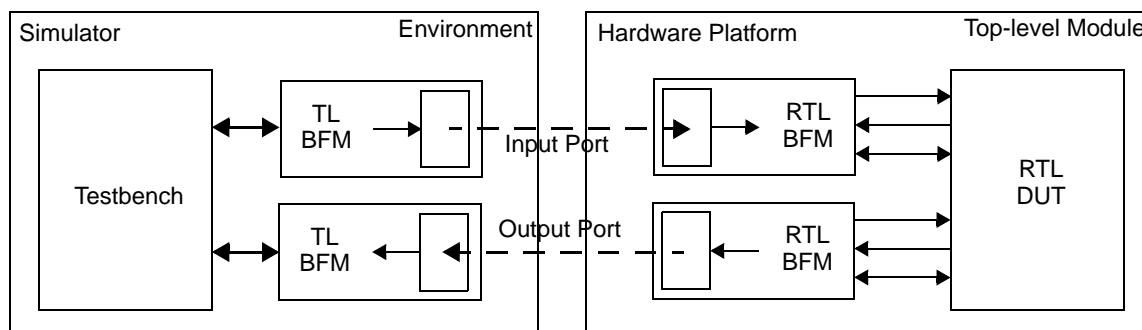
# 1

## 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_SCEMI	SceMi-compliant platform (not yet supported).

### *Example 1-2 Selecting target platform*

```
+define+VMM_HW_ARCH_NULL ...
```

---

## Hardware-Assisted Portion

When compiling the hardware-assisted portion of the design and testbench for the target platform, the `VMM_HW_SYNTHESIS_ON` symbol must be defined in addition to the target platform selection symbol. When this symbol is defined, only synthesizable Verilog-1995 compliant code is supplied by the file `vmm_hw_rtl.sv`.

### *Example 1-3 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 "vmm\_hw\_arch\_null" . 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 platform, 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 1-4 Selecting target platform access class*

```
vmm_hw_arch platform;  
  
`ifdef VMM_HW_ARCH_NULL  
    vmm_hw_arch_null arch = new();  
`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 file `vmm_hw.sv`.

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

*Example 1-5 Compiling simulated portion*

```
+define+VMM_HW_ARCH_SCEMI \  
bfm.v tb_top.v tb_env.sv ...
```



# 2

## Implementing HW-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. 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. How the 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 put 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 run-time 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 cycles.

## **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 bytes 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 bytes 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
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 layer 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, or data extracted from a local memory. Similarly, different checking strategies can be used: computing then checking a checksum, or 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. It is therefore odd that such a message is required.

Remember that the goal is to create a verification environment that will be portable, as-is, to different hardware-assistance platform, including a purely simulated one. 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. But 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.

**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, that the transactors be implemented using properly synthesizable RTL code is an implicit requirement.

**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 transactors 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.

### *Example 2-4 Message interface instances*

```
module hw_reset_bfm(rst);
    output rst;
    ...
    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. The controlled clock signals, provided by “vmm\_hw\_clock” instances, are reserved for the DUT. 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 posegde 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 an "protocol stall" message.**

This guideline is a corollary of the previous guideline. Stalling the protocol should be treated as just another transaction the transactor can do. 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 transactors 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.

**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 1-4](#). 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](#).

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.

### *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 23](#)

- [vmm\\_hw\\_out\\_if](#) ..... page 25
- [vmm\\_hw\\_clock](#) ..... page 27
- [vmm\\_hw\\_clock\\_control](#) ..... page 29
- [vmm\\_hw\\_arch](#) ..... page 31
- [vmm\\_hw\\_arch\\_null](#) ..... page 38
- [vmm\\_hw\\_in\\_port](#) ..... page 39
- [vmm\\_hw\\_out\\_port](#) ..... page 43

## vmm\_hw\_in\_if

Synthesizable end-point 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.

## Examples

### *Example A-1*

TBD

## vmm\_hw\_out\_if

Synthesizable end-point 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.

## Examples

### *Example A-2*

TBD

## 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 thus 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.

## Examples

### *Example A-3*

TBD

# 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 thus 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.

## Examples

### *Example A-4*

TBD

## vmm\_hw\_arch

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

### Platform Access Classes

- [vmm\\_hw\\_arch\\_null](#) ..... page 38
- [vmm\\_hw\\_arch::clk\\_control\(\)](#) ..... page 32
- [vmm\\_hw\\_arch::create\\_in\\_port\(\)](#) ..... page 33
- [vmm\\_hw\\_arch::create\\_out\\_port\(\)](#) ..... page 35
- [vmm\\_hw\\_arch::init\\_sim\(\)](#) ..... page 37

## 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);
```

### 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

an absolute hierarchical reference to the “vmm\_hw\_clock” module instance controlled by the specified clock control module.

ctl

an absolute hierarchical reference to a “vmm\_hw\_clock\_control” module instance that controls the specified clock source.

### Examples

#### *Example A-5*

TBD

## 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 = "");
```

### 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

an absolute hierarchical reference to the “vmm\_hw\_in\_if” module instance corresponding to the platform-side endpoint.

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.

## Examples

### *Example A-6*

TBD

## 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 = "");
```

### 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

an absolute hierarchical reference to the “vmm\_hw\_out\_if” module instance corresponding to the platform-side endpoint.

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.

## Examples

### *Example A-7*

TBD

## **vmm\_hw\_arch::init\_sim()**

Initialize the platform to get ready for simulation.

### **SystemVerilog**

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

### **Description**

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

### **Examples**

#### *Example A-8*

TBD

## vmm\_hw\_arch\_null

Simulation-only pseudo-platform access class.

### SystemVerilog

```
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.

### Examples

#### *Example A-9*

TBD

## 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 40
- [vmm\\_hw\\_in\\_port::wait\\_is\\_rdy\(\)](#) ..... page 41
- [vmm\\_hw\\_in\\_port::send\(\)](#) ..... page 42

## **vmm\_hw\_in\_port::is\_rdy()**

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

### **SystemVerilog**

```
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.

### **Examples**

#### *Example A-10*

TBD

## **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();
```

### **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.

### **Examples**

#### *Example A-11*

TBD

## **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);
```

### **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.

### **Examples**

#### *Example A-12*

TBD

## 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 44](#)
- [vmm\\_hw\\_out\\_port::wait\\_is\\_rdy\(\)](#) ..... [page 45](#)
- [vmm\\_hw\\_out\\_port::receive\(\)](#) ..... [page 46](#)

## **vmm\_hw\_out\_port::is\_rdy()**

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

### **SystemVerilog**

```
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.

### **Examples**

#### *Example A-13*

TBD

## **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();
```

### **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.

### **Examples**

#### *Example A-14*

TBD

## 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);
```

### 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

The uncontrolled clock cycle count when the message was transferred through the “[vmm\\_hw\\_out\\_if](#)” module instance.

### Examples

#### *Example A-15*

TBD

# B

## Platform Integration

---

This appendix details how to integrate a new platform in the VMM hardware abstraction layer package. This appendix 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"` and `"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](#).

## Component Summary

- [vmm\\_hw\\_arch\\_XYZ](#) ..... page 49
- [vmm\\_hw\\_in\\_port\\_XYZ](#) ..... page 59
- [vmm\\_hw\\_out\\_port\\_XYZ](#) ..... page 63

## 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 mechanism must be encapsulated within this class. 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 50
- [vmm\\_hw\\_arch\\_XYZ::create\\_in\\_port\(\)](#) ..... page 52
- [vmm\\_hw\\_arch\\_XYZ::create\\_out\\_port\(\)](#) ..... page 54
- [vmm\\_hw\\_arch::connect\\_to\(\)](#) ..... page 56
- [vmm\\_hw\\_arch::init\\_sim\(\)](#) ..... page 58

## 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);
```

### 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.

## Examples

### *Example B-2*

TBD

## 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 = "");
```

### 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

absolute hierarchical reference to the “vmm\_hw\_in\_if” module instance corresponding to the platform-side message interface, suffixed with “.vitf”.

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 withing 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:

```
this.connect_to(itf.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.

## Examples

### *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 = "");
```

### 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

an absolute hierarchical reference to the “vmm\_hw\_out\_if” module instance corresponding to the platform-side message interface, suffixed with “.vitf”.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 withing 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:

```
this.connect_to(itf.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.

## Examples

### *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);
```

### **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.

## Examples

### *Example B-5*

TBD

## **vmm\_hw\_arch::init\_sim()**

Initialize the hardware assistance platform.

### **SystemVerilog**

```
virtual function void 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.

### **Examples**

#### *Example B-6*

TBD

## 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

*Example B-7*

TBD

### Summary

- [vmm\\_hw\\_in\\_port\\_XYZ::is\\_rdy\(\)](#) ..... page 60
- [vmm\\_hw\\_in\\_port\\_XYZ::wait\\_is\\_rdy\(\)](#) ..... page 61
- [vmm\\_hw\\_in\\_port\\_XYZ::send\(\)](#) ..... page 62

## **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();
```

### **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.

### **Examples**

#### *Example B-8*

TBD

## **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();
```

### **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.

### **Examples**

#### *Example B-9*

TBD

## **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);
```

### **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.

### **Examples**

#### *Example B-10*

TBD

# 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 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\\_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 need to be passed from the “[vmm\\_hw\\_arch\\_XYZ](#)” class.

All documented virtual methods must be implemented

## Example

### *Example B-11*

TBD

## Summary

- [vmm\\_hw\\_out\\_port\\_XYZ::is\\_rdy\(\)](#) ..... page 64
- [vmm\\_hw\\_out\\_port\\_XYZ::wait\\_is\\_rdy\(\)](#) ..... page 65
- [vmm\\_hw\\_out\\_port\\_XYZ::receive\(\)](#) ..... page 66

## **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();
```

### **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.

### **Examples**

#### *Example B-12*

TBD

## **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();
```

### **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.

### **Examples**

#### *Example B-13*

TBD

## **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);
```

### **DATA 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.

### **Examples**

#### *Example B-14*

TBD