



Verification Academy



# Cookbook

Online Methodology Documentation from the  
Mentor Graphics Verification Methodology Team

Contact [VMDOC@mentor.com](mailto:VMDOC@mentor.com)

<http://verificationacademy.com>

# Table of Contents

## Articles

<b>Introduction</b>	<b>0</b>
Cookbook/Introduction	0
Cookbook/Acknowledgements	1
<b>Testbench Architecture</b>	<b>2</b>
Testbench	2
Testbench/Build	9
Testbench/Blocklevel	19
Testbench/IntegrationLevel	29
Component	39
Agent	42
Phasing	48
Factory	53
UsingFactoryOverrides	56
SystemVerilogPackages	62
<b>Connections to DUT Interfaces</b>	<b>65</b>
Connections	65
SVCreationOrder	71
Connect/SystemVerilogTechniques	73
ParameterizedTests	75
Connect/Virtual Interface	78
Config/VirtInterfaceConfigDb	86
Connect/VirtInterfacePackage	90
Connect/VirtInterfaceConfigPkg	93
Connect/TwoKingdomsFactory	97
DualTop	103
VirtInterfaceFunctionCallChain	106
BusFunctionalModels	108
ProtocolModules	111
Connect/AbstractConcrete	115

Connect/AbstractConcreteConfigDB	118
<b>Configuring a Test Environment</b>	<b>126</b>
Configuration	126
Resources/config db	131
Config/Params Package	134
Config/ConfiguringSequences	139
ResourceAccessForSequences	142
MacroCostBenefit	145
<b>Analysis Components &amp; Techniques</b>	<b>146</b>
Analysis	146
AnalysisPort	149
AnalysisConnections	152
MonitorComponent	158
Predictors	161
Scoreboards	163
MetricAnalyzers	170
PostRunPhases	172
Matlab/Integration	175
<b>End Of Test Mechanisms</b>	<b>183</b>
EndOfTest	183
Objections	185
<b>Sequences</b>	<b>188</b>
Sequences	188
Sequences/Items	193
Transaction/Methods	195
Sequences/API	200
Connect/Sequencer	204
Driver/Sequence API	206
Sequences/Generation	213
Sequences/Overrides	221
Sequences/Virtual	223
Sequences/VirtualSequencer	231

Sequences/Hierarchy	237
Sequences/SequenceLibrary	242
Driver/Use Models	246
Driver/Unidirectional	247
Driver/Bidirectional	250
Driver/Pipelined	255
Sequences/Arbitration	267
Sequences/Priority	276
Sequences/LockGrab	277
Sequences/Slave	284
Stimulus/Signal Wait	290
Stimulus/Interrupts	294
Sequences/Stopping	301
Sequences/Layering	302
<b>Register Abstraction Layer</b>	<b>308</b>
Registers	308
Registers/Specification	315
Registers/Adapter	317
Registers/Integrating	321
Registers/Integration	327
Registers/RegisterModelOverview	332
Registers/ModelStructure	334
Registers/QuirkyRegisters	344
Registers/ModelCoverage	349
Registers/BackdoorAccess	354
Registers/Generation	357
Registers/StimulusAbstraction	358
Registers/MemoryStimulus	370
Registers/SequenceExamples	375
Registers/BuiltInSequences	382
Registers/Configuration	386
Registers/Scoreboarding	389
Registers/FunctionalCoverage	395

<b>Testbench Acceleration through Co-Emulation</b>	<b>401</b>
Emulation	401
Emulation/SeparateTopLevels	404
Emulation/SplitTransactors	410
Emulation/BackPointers	415
Emulation/DefiningAPI	419
Emulation/Example	422
Emulation/Example/APBDriver	430
Emulation/Example/SPIAgent	435
Emulation/Example/TopLevel	441
<b>Debug of SV and UVM</b>	<b>444</b>
BuiltInDebug	444
Reporting/Verbosity	455
UVM/CommandLineProcessor	460
<b>UVM Connect - SV-SystemC interoperability</b>	<b>464</b>
UvmConnect	464
UvmConnect/Connections	466
UvmConnect/Conversion	468
UvmConnect/CommandAPI	472
<b>UVM Express - step by step improvement</b>	<b>476</b>
UvmExpress	476
UvmExpress/DUT	481
UvmExpress/BFM	485
UvmExpress/WritingBfmTests	490
UvmExpress/FunctionalCoverage	498
UvmExpress/ConstrainedRandom	503
<b>Appendix - Deployment</b>	<b>516</b>
OVM2UVM	516
OVM2UVM/DeprecatedCode	527
OVM2UVM/SequenceLibrary	528
OVM2UVM/Phasing	530

OVM2UVM/ConvertPhaseMethods	535
UVC/UvmVerificationComponent	537
Package/Organization	548
<b>Appendix - Coding Guidelines</b>	<b>555</b>
SV/Guidelines	555
UVM/Guidelines	569
<b>Appendix - Glossary of Terms</b>	<b>579</b>
Doc/Glossary	579

## Datestamp:

- This document is a snapshot of dynamic content from the Online Methodology Cookbook
- Created from <http://verificationacademy.com/uvm-ovm> on Wed, 04 Sep 2013 09:48:38 UTC

---

# Introduction

---

## Cookbook/Introduction

---

### **Universal Verification Methodology (UVM)**

The Accellera UVM standard was built on the principle of cooperation between EDA vendors and customers; this was made possible by the strong foundation of knowledge and experience that was donated to the standardization effort in the form of the existing OVM code base and contributions from VMM.

The result is a hybrid of technologies that originated in Mentor's AVM, Mentor & Cadence's OVM, Verisity's eRM, and Synopsys's VMM-RAL, tried and tested with our respective customers, along with several new technologies such as Resources, TLM2 and Phasing, all developed by Mentor and others to form UVM as we know it.

Combined, these features provide a powerful, flexible technology and methodology to help you create scalable, reusable, and interoperable testbenches. With the OVM at its core, the UVM already embodies years of object-oriented design and methodology experience, all of which can be applied immediately to a UVM project.

When we commenced work on UVM, Mentor set out to capture documentation of our existing OVM methodology at a fine level of granularity. In the process, we realized that learning a new library and methodology needed to be a dynamic and interactive experience, preferably consumed in small, easily digested spoonfuls. To reinforce each UVM and OVM concept or best practice, we developed many realistic, focused code examples. The end result is the UVM/OVM Online Methodology Cookbook, whose recipes can be adapted and applied in many different ways by our field experts, customers, and partners alike.

The book you are holding contains excerpts from this online resource, covering many aspects of the UVM and OVM. Check out our UVM website to learn much more, and join others in finding out how you can leverage the UVM in your specific applications.

**Find us online at** <http://verificationacademy.com/cookbook>"

# Cookbook/Acknowledgements

---

## **UVM/OVM Cookbook Authors:**

- Gordon Allan
- Mike Baird
- Rich Edelman
- Adam Erickson
- Michael Horn
- Mark Peryer
- Adam Rose
- Kurt Schwartz

We acknowledge the valuable contributions of all our extended team of contributors and reviewers, and those who help deploy our methodology ideas to our customers, including: Alain Gonier, Allan Crone, Bahaa Osman, Dave Rich, Eric Horton, Gehan Mostafa, Graeme Jessiman, Hans van der Schoot, Hager Fathy, Jennifer Adams, John Carroll, John Amouroux, Jason Polychronopoulos, John Stickley, Nigel Elliot, Peet James, Ray Salemi, Shashi Bhutada, Tim Corcoran, and Tom Fitzpatrick.

# Testbench Architecture

## Testbench

This chapter covers the basics and details of UVM testbench architecture, construction, and leads into other chapters covering each of the constituent parts of a typical UVM testbench.

### Testbench Chapter contents:

- Testbench (this page) - top-level introduction into testbench architecture UVM-style
- Testbench/Build - testbench hierarchy construction in the UVM 'build()' phase
- Testbench/Blocklevel - architecture of a unit-level UVM test environment
- Testbench/IntegrationLevel - example architecture of vertical reuse testbench
- Agent - architecture of a single interface agent
- UVM Phases - execution phases in an UVM testbench component
- UVM Factory - machinery for manufacture of configurable objects

## Topic Overview

### How an UVM testbench differs from a traditional module based testbench

In Verilog or VHDL, a testbench consists of a hierarchy of modules containing testbench code that are connected to the design under test (DUT). The modules contain stimulus and response checking code which is loaded into simulator memory along with the DUT at the beginning of the simulation and is present for the duration of the simulation. Therefore, the classic Verilog testbench wrapped around a DUT consists of what are known as static objects.

SystemVerilog builds on top of Verilog by adding abstract language constructs targeted at helping the verification process. One of the key additions to the language was the class. SystemVerilog classes allow Object Oriented Programming (OOP) techniques to be applied to testbenches. The UVM itself is a library of base classes which facilitate the creation of structured testbenches using code which is open source and can be run on any SystemVerilog IEEE 1800 simulator.

Like classes in any other OOP language such as C++ and Java, SystemVerilog class definitions are templates for an object that is constructed in memory. Once created, that object persists in memory until it is de-referenced and garbage collected by an automatic background process. The class template defines the members of the class which can either be data variables or methods. In SystemVerilog, the methods can either be functions which are non-time consuming, or tasks which can consume time. Since a class object has to be constructed before it exists in memory the creation of a class hierarchy in a SystemVerilog testbench has to be initiated from a module since a module is a static object that is present at the beginning of the simulation. For the same reason, a class cannot contain a module. Classes are referred to as dynamic objects because they can come and go during the life time of a simulation.

```
//  
// Example to show how a class is constructed from within a static object (a module)
```

```
//  
  
//  
// Example class that contains a message and some convenience methods  
//  
class example;  
  
string message;  
  
function void set_message(string ip_string);  
    message = ip_string;  
endfunction: set_message  
  
function void print();  
    $display("%s", message);  
endfunction: print  
  
endclass: example  
  
//  
// Module that uses the class - class is constructed, used and dereferenced  
// in the initial block, after the simulation starts  
//  
module tb;  
  
example C; // Null handle after elaboration  
  
initial begin  
    C = new(); // Handle points to C object in memory  
    C.set_message("This object has been created");  
    #10;  
    C.print();  
    C = null; // C has been dereferenced, object can be garbage collected  
end  
  
endmodule: tb
```

## The UVM Package

The UVM package contains a class library that comprises three main types of classes, uvm\_components which are used to construct a class based hierarchical testbench structure, uvm\_objects which are used as data structures for configuration of the testbench and uvm\_transactions which are used in stimulus generation and analysis.

An UVM testbench will always have a top level module which contains the DUT and the testbench connections to it. The process of connecting a DUT to an UVM class based testbench is described in the article on DUT- testbench connections.

The top level module will also contain an initial block which will contain a call to the UVM run\_test() method. This method starts the execution of the UVM phases, which controls the order in which the testbench is built, stimulus is generated and then reports on the results of the simulation.

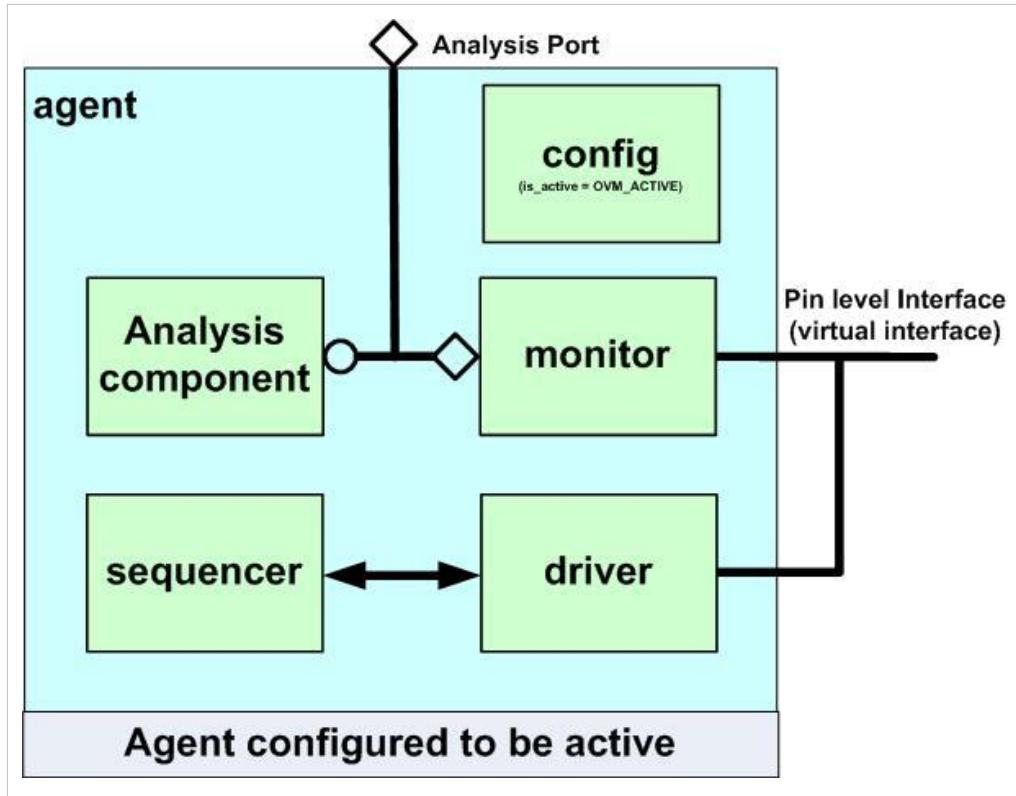
## UVM Testbench Hierarchy

UVM testbenches are built from classes derived from the uvm\_component base class. The testbench hierarchy is determined by a series of 'has-a' class relationships, in other words which components contain which other components. The top level class in an UVM testbench is usually known as the test class and this class is responsible for configuring the testbench, initiating the construction process by building the next level down in the hierarchy and by initiating the stimulus by starting the main sequence. For a given verification environment, the testbench hierarchy below the test class is reasonably consistent, and each test case is implemented by extending a test base class.

The UVM testbench architecture is modular to facilitate the reuse of groups of verification components either in different projects (horizontal reuse) or at a higher level of integration in the same project (vertical reuse). There are two main collective component types used to enable reuse - the env (short for environment) and the agent.

## The Agent

Most DUTs have a number of different signal interfaces, each of which have their own protocol. The UVM agent collects together a group of uvm\_components focused around a specific pin-level interface. The purpose of the agent is to provide a verification component which allows users to generate and monitor pin level transactions. A SystemVerilog package is used to gather all the classes associated with an agent together into one namespace.



The contents of an agent package will usually include:

- **A Sequence\_item** - The agent will have one or more sequence items which are used to either define what pin level activity will be generated by the agent or to report on what pin level activity has been observed by the agent.
- **A Driver** - The driver is responsible for converting the data inside a series of sequence\_items into pin level transactions.
- **A Sequencer** - The role of the sequencer is to route sequence\_items from a sequence where they are generated to/from a driver.
- **A Monitor** - The monitor observes pin level activity and converts its observations into sequence\_items which are sent to components such as scoreboards which use them to analyse what is happening in the testbench.
- **Configuration object** - A container object, used to pass information to the agent which affects what it does and how it is built and connected.

Each agent should have a configuration object, this will contain a reference to the virtual interface which the driver and the monitor use to access pin level signals. The configuration object will also contain other data members which will control which of the agents sub-components are built, and it may also contain information that affects the behaviour of the agents components (e.g. error injection, or support for a protocol variant)

The agent configuration object contains an active bit which can be used to select whether the agent is passive - i.e. the driver and sequencer are not required, or active. It may also contain other fields which control whether other

sub-component classes such as functional coverage monitors or scoreboards get built or not.

Other classes that might be included in an agent package:

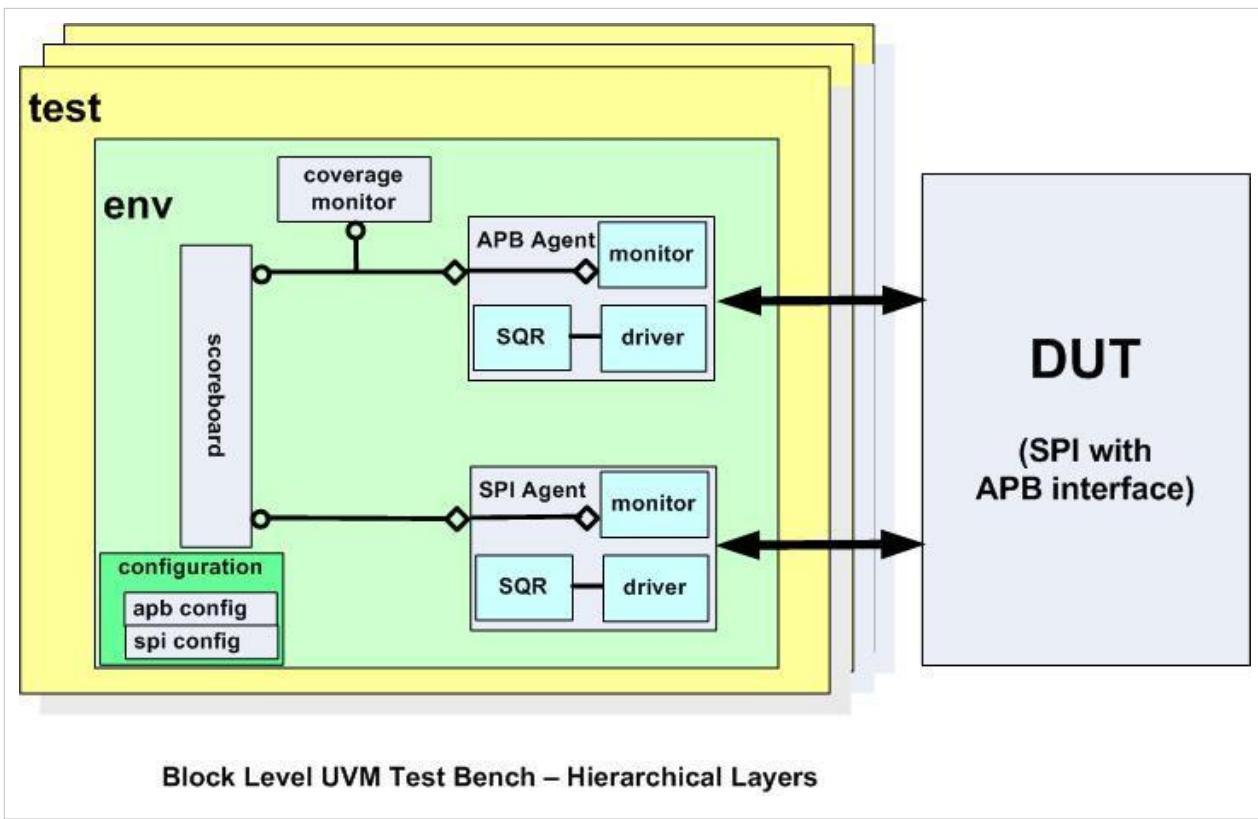
- **Functional coverage monitor** - to collect protocol specific functional coverage information
- **Scoreboard** - usually of limited use
- **A responder** - A driver that responds to bus events rather than creating them (i.e. a slave version of the driver rather than a master version).
- **(API) Sequences** - Utility sequences likely to be of general use, often implementing an API layer for the driver.

## The Env

The environment, or env, is a container component for grouping together sub-components orientated around a block, or around a collection of blocks at higher levels of integration.

### Block Level Env

In a block level UVM testbench, the environment (env) is used to collect together the agents needed to communicate with the DUT's interfaces together in one place. Like the agent, the different classes associated with the env are organized into a SystemVerilog package, which will import the agent packages. In addition to the agents, the env will also contain some or all of the following types of components:



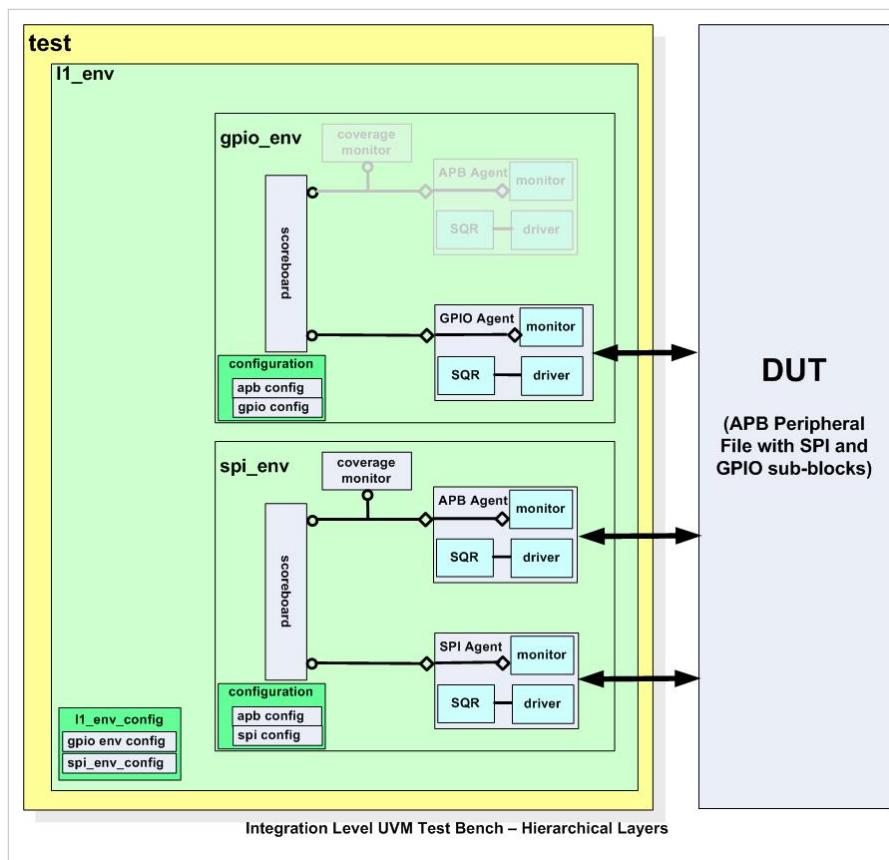
- **Configuration object** - The env will have a configuration object that enables the test writer to control which of the environments sub-components are built. The env configuration object should also contain a handle for the configuration object for each of the agents that it contains. These are then assigned to the envs agents using `set_config_object`.

- **Scoreboards** - A scoreboard is an analysis component that checks that the DUT is behaving correctly. UVM scoreboards use analysis transactions from the monitors implemented inside agents. A scoreboard will usually compare transactions from at least two agents, which is why it is usually present in the env.
- **Predictors** - A predictor is a component that computes the response expected from the stimulus, it is generally used in conjunction with other components such as the scoreboard.
- **Functional Coverage Monitors** - A functional coverage monitor analysis component contains one or more covergroups which are used to gather functional coverage information related to what has happened in a testbench during a test case. A functional coverage monitor is usually specific to a DUT.

The diagram shows a block level testbench which consists of a series of tests which build an env which contains several analysis components and two agents.

### Integration Level Env

When blocks are integrated to create a sub-system, vertical reuse can be achieved by reusing the envs used in each of the block level testbenches merged together into a higher level env. The block level envs provide all of the structures required to test each block, but as a result of the integration process, not all the block level interfaces are exposed at the boundary and so some of the functionality of the block level envs will be redundant. The integration level env then needs to be configured to make agents connected to internal interfaces passive, or possibly even not to include an agent at all. This configuration is done in the test, and the configuration object for each sub-env is nested inside the configuration object for the env at the next level of hierarchy.



As an illustration, the diagram shows a first level of integration where two block level environments have been merged together to test the peripheral file. The 'greyed out' components in the envs are components that are no longer used in the

integration level environment. The configuration object for the integration level contains the rest of the configuration objects nested inside it.

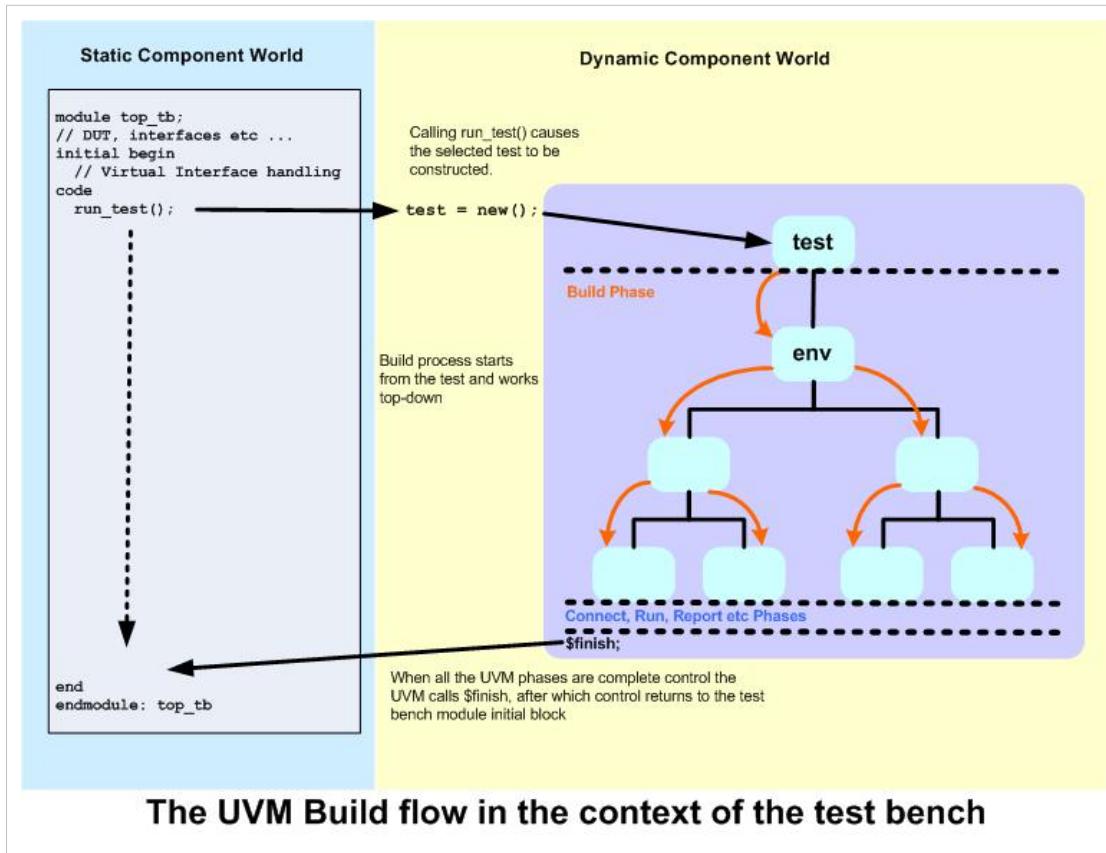
Further levels of integration can be accommodated by layering multiple integration level envs inside each other.

## **The UVM Testbench Build and Connection Process**

Before an UVM testbench can start to apply stimulus, its component hierarchy has to be built and the connections between the verification components has to be made. The process for configuring and building agents as well as block level and integration level UVM testbenches is described in the article on building UVM testbenches.

# Testbench/Build

The first phase of an UVM testbench is the build phase. During this phase the uvm\_component classes that make up the testbench hierarchy are constructed into objects. The construction process works top-down with each level of the hierarchy being constructed before the next level is configured and constructed. This approach to construction is referred to as deferred construction.



The UVM testbench is activated when the `run_test()` method is called in an initial block in the top level test module. This method is an UVM static method, and it takes a string argument that defines the test to be run and constructs it via the factory. Then the UVM infrastructure starts the build phase by calling the test classes `build` method.

During the execution of the tests build phase, the testbench component configuration objects are prepared and assignments to the testbench module interfaces are made to the virtual interface handles in the configuration objects. The next step is for the configuration objects to be put into the test's configuration table. Finally the next level of hierarchy is built.

At the next level of hierarchy, the configuration object prepared by the test is retrieved and further configuration may take place. Before the configuration object is used to guide the configuration and conditional construction of the next level of hierarchy it could be modified by that level of hierarchy. This conditional construction affects the topology or hierarchical structure of the testbench.

The build phase works top-down and so the process is repeated for each successive level of the testbench hierarchy until the bottom of the hierarchical tree is reached.

After the build phase has completed, the connect phase is used to ensure that all intra-component connections are made. The connect phase works from the bottom to the top of the testbench hierarchy. Following the connect phase, the rest of the UVM phases run to completion before control is passed back to the testbench module.

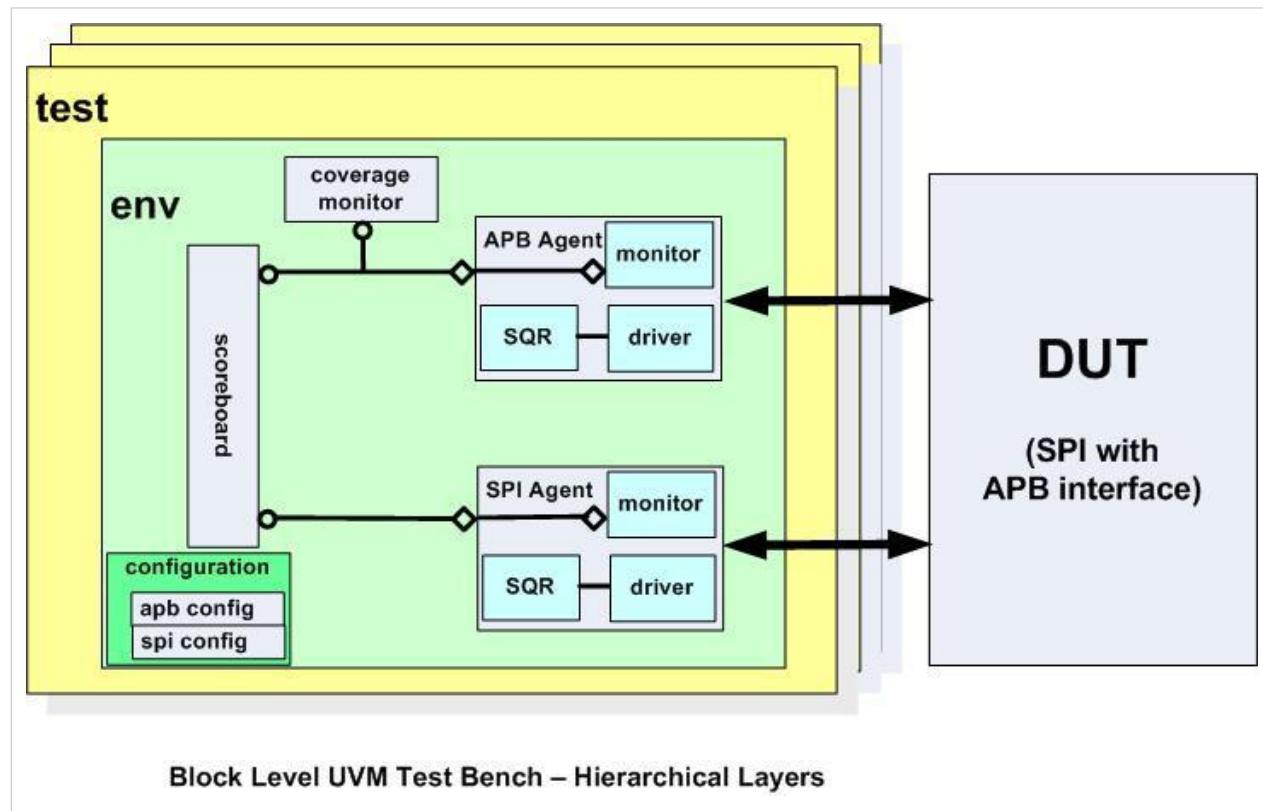
## The Test Is The Starting Point For The Build Process

The build process for an UVM testbench starts from the test class and works top-down. The test class build method is the first to be called during the build phase and what the test method sets up determines what gets built in an UVM testbench. The function of the tests build method is to:

- Set up any factory overrides so that configuration objects or component objects are created as derived types
- Create and configure the configuration objects required by the various sub-components
- Assign any virtual interface handles put into configuration space by the testbench module
- Build up a nested env configuration object which is then set into the configuration space
- Build the next level down in the testbench hierarchy, usually the top-level env

For a given design verification environment most of the work done in the build method will be the same for all the tests, so it is recommended that a test base class is created which can be easily extended for each of the test cases.

To help explain how the test build process works, a block level verification environment will be referred to. This example is an environment for an SPI master interface DUT and it contains two agents, one for its APB bus interface and one for its SPI interface. A detailed account of the build and connect processes for this example can be found in the Block Level Testbench Example article.



## Factory Overrides

The UVM factory allows an UVM class to be substituted with another derived class at the point of construction. This facility can be useful for changing or updating component behaviour or for extending a configuration object. The factory override must be specified before the target object is constructed, so it is convenient to do it at the start of the build process.

## Sub-Component Configuration Objects

Each collective component such as an agent or an env should have a configuration object which defines their structure and behavior. These configuration objects should be created in the test build method and configured according to the requirements of the test case. If the configuration of the sub-component is either complex or is likely to change then it is worth adding a virtual function call to take care of the configuration, since this can be overloaded in test cases extending from the base test class.

```
//  
// Class Description:  
//  
//  
class spi_test_base extends uvm_test;  
  
// UVM Factory Registration Macro  
//  
`uvm_component_utils(spi_test_base)  
  
//-----  
// Data Members  
//-----  
  
//-----  
// Component Members  
//-----  
  
// The environment class  
spi_env m_env;  
// Configuration objects  
spi_env_config m_env_cfg;  
apb_agent_config m_apb_cfg;  
spi_agent_config m_spi_cfg;  
  
//-----  
// Methods  
//-----  
  
// Standard UVM Methods:  
extern function new(string name = "spi_test_base", uvm_component parent = null);  
extern function void build_phase( uvm_phase phase );
```

```
extern virtual function void configure_env(spi_env_config cfg);
extern virtual function void configure_apb_agent(apb_agent_config cfg);

endclass: spi_test_base

function spi_test_base::new(string name = "spi_test_base", uvm_component parent = null);
  super.new(name, parent);
endfunction

// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
  // Create env configuration object
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // Call function to configure the env
  configure_env(m_env_cfg);
  // Create apb agent configuration object
  m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
  // Call function to configure the apb_agent
  configure_apb_agent(m_apb_cfg);
  // More to follow
endfunction: build_phase

// 
// Convenience function to configure the env
//
// This can be overloaded by extensions to this base class
function void spi_test_base::configure_env(spi_env_config cfg);
  cfg.has_functional_coverage = 1;
  cfg.has_reg_scoreboard = 0;
  cfg.has_spi_scoreboard = 1;
endfunction: configure_env

// 
// Convenience function to configure the apb agent
//
// This can be overloaded by extensions to this base class
function void spi_test_base::configure_apb_agent(apb_agent_config cfg);
  cfg.active = UVM_ACTIVE;
  cfg.has_functional_coverage = 0;
  cfg.has_scoreboard = 0;
endfunction: configure_apb_agent
```

## Assigning Virtual Interfaces From The Configuration Space

Before the UVM run\_test() method is called, the links to the signals on the top level I/O boundary of the DUT should have been made by connecting them to SystemVerilog interfaces and then a handle to each interface should have been assigned to a virtual interface handle which is passed in to the test using uvm\_config\_db::set. See the article on virtual interfaces for more information on this topic.

In the test's build method, these virtual interface references need to be assigned to the virtual interface handles inside the relevant component configuration objects. Then, individual components access the virtual interface handle inside their configuration object in order to drive or monitor DUT signals. In order to keep components modular and reusable, drivers and monitors should not get their virtual interface pointers directly from the configuration space, only from their configuration object. The test class is the correct place to ensure that the virtual interfaces are assigned to the right verification components via their configuration objects.

The following code shows how the SPI testbench example uses the uvm\_config\_db::get method to make virtual interface assignments to the virtual interface handle in the apb\_agents configuration object:

```
// The build method from before, adding the apb agent virtual interface assignment
// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
  // Create env configuration object
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // Call function to configure the env
  configure_env(m_env_cfg);
  // Create apb agent configuration object
  m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
  // Call function to configure the apb_agent
  configure_apb_agent(m_apb_cfg);
  // Adding the apb virtual interface:
  if( !uvm_config_db #( virtual apb3_if )::get(this, "", "APB_vif",m_apb_cfg.APB) ) `uvm_error(...)

  // More to follow
endfunction: build_phase
```

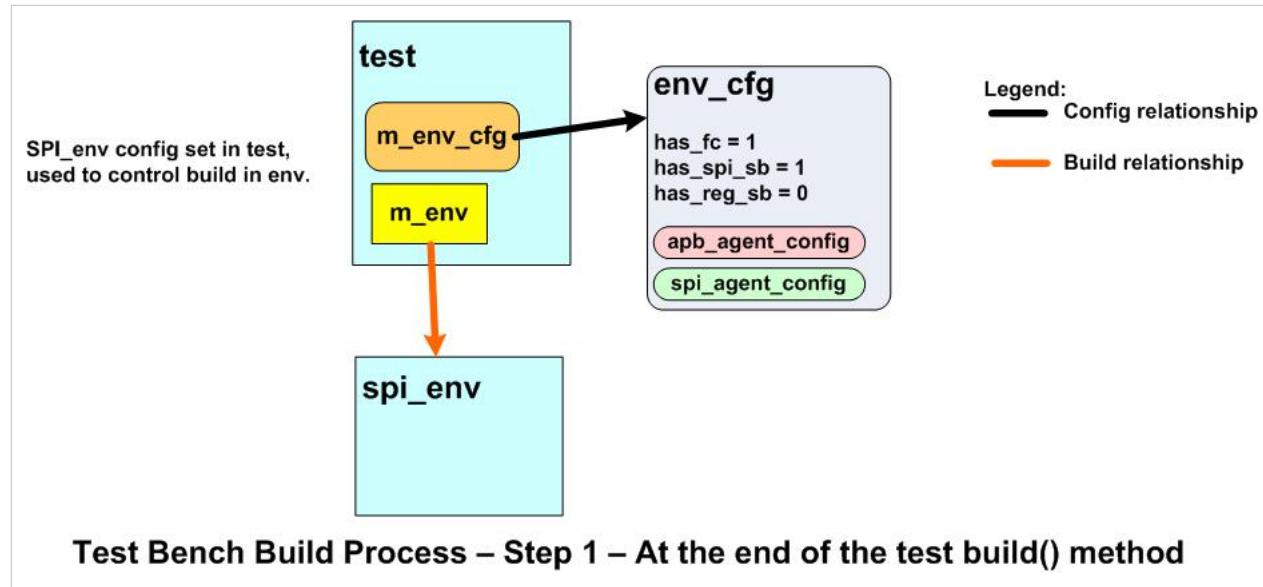
## Nesting Sub-Component Configuration Objects

Configuration objects are passed to sub-components via the UVM component configuration space from the test. They can be passed individually, using the path argument in the uvm\_config\_db::set method to control which components can access the objects. However a common requirement is that intermediate components also need to do some local configuration.

Therefore, an effective way to approach the passing of configuration objects through a testbench hierarchy is to nest the configuration objects inside each other in a way that reflects the hierarchy itself. At each intermediate level in the testbench, the configuration object for that level is unpacked and then its sub-configuration objects are re-configured (if necessary) and then passed to the relevant components using uvm\_config\_db::set.

Following the SPI block level environment example, each of the agents will have a separate configuration object. The envs configuration object will have a handle for each of the agents configuration object. In the test, all of the configuration objects will be constructed and configured from the test case viewpoint, then the agent configuration object

handles inside the env configuration object will be assigned to the actual agent configuration objects. Then the env configuration object would be set into the configuration space, to be retrieved when the env is built.



For more complex environments, additional levels of nesting will be required.

```

//  

// Configuration object for the spi_env:  

//  

//  

//  

// Class Description:  

//  

//  

class spi_env_config extends uvm_object;  

// UVM Factory Registration Macro  

//  

`uvm_object_utils(spi_env_config)  

//-----  

// Data Members  

//-----  

// Whether env analysis components are used:  

bit has_functional_coverage = 1;  

bit has_reg_scoreboard = 0;  

bit has_spi_scoreboard = 1;  

// Configurations for the sub_components

```

```
apb_config m_apb_agent_cfg;
spi_agent_config m_spi_agent_cfg;

//-----
// Methods
//-----

// Standard UVM Methods:
extern function new(string name = "spi_env_config");

endclass: spi_env_config

function spi_env_config::new(string name = "spi_env_config");
  super.new(name);
endfunction

// 
// Inside the spi_test_base class, the agent config handles are assigned:
// 
// The build method from before, adding the apb agent virtual interface assignment
// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
  // Create env configuration object
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // Call function to configure the env
  configure_env(m_env_cfg);
  // Create apb agent configuration object
  m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
  // Call function to configure the apb_agent
  configure_apb_agent(m_apb_cfg);
  // Adding the apb virtual interface:
  if( !uvm_config_db #( virtual apb3_if )::get(this, "", "APB_vif",m_apb_cfg.APB) ) `uvm_error(...)

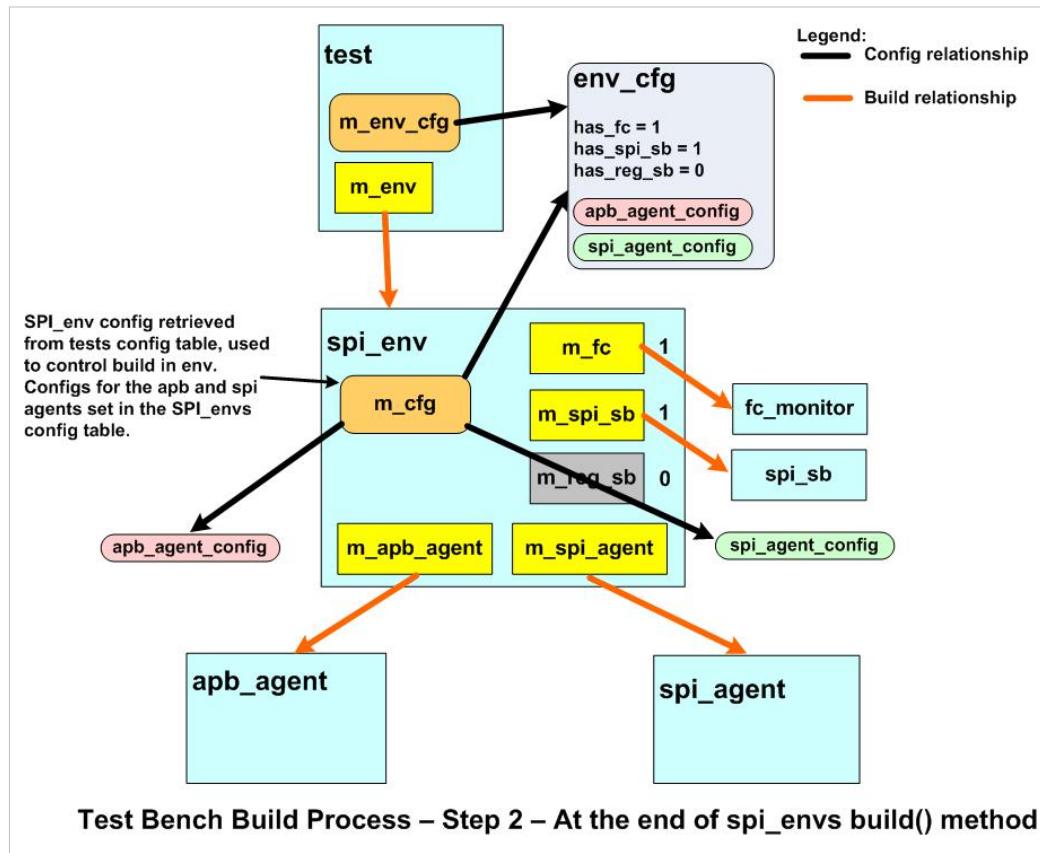
  // Assign the apb_agent config handle inside the env_config:
  m_env_cfg.m_apb_agent_cfg = m_apb_cfg;
  // Repeated for the spi configuration object
  m_spi_cfg = spi_agent_config::type_id::create("m_spi_cfg");
  configure_spi_agent(m_spi_cfg);
  if( !uvm_config_db #( virtual apb3_if )::get(this, "", "SPIvif",m_spi_cfg.SPI) ) `uvm_error(...)

  m_env_cfg.m_spi_agent_cfg = m_spi_cfg;
  // Now env config is complete set it into config space:
  uvm_config_db #( spi_env_config )::set( this , "*m_spi_agent*", "spi_env_config", m_env_cfg );
  // Now we are ready to build the spi_env:
  m_env = spi_env::type_id::create("m_env", this);
```

```
endfunction: build_phase
```

## Building The Next Level Of Hierarchy

The final stage of the test build process is to build the next level of testbench hierarchy using the UVM factory. This usually means building the top level env, but there could be more than one env or there could be a conditional build with a choice being made between several envs.

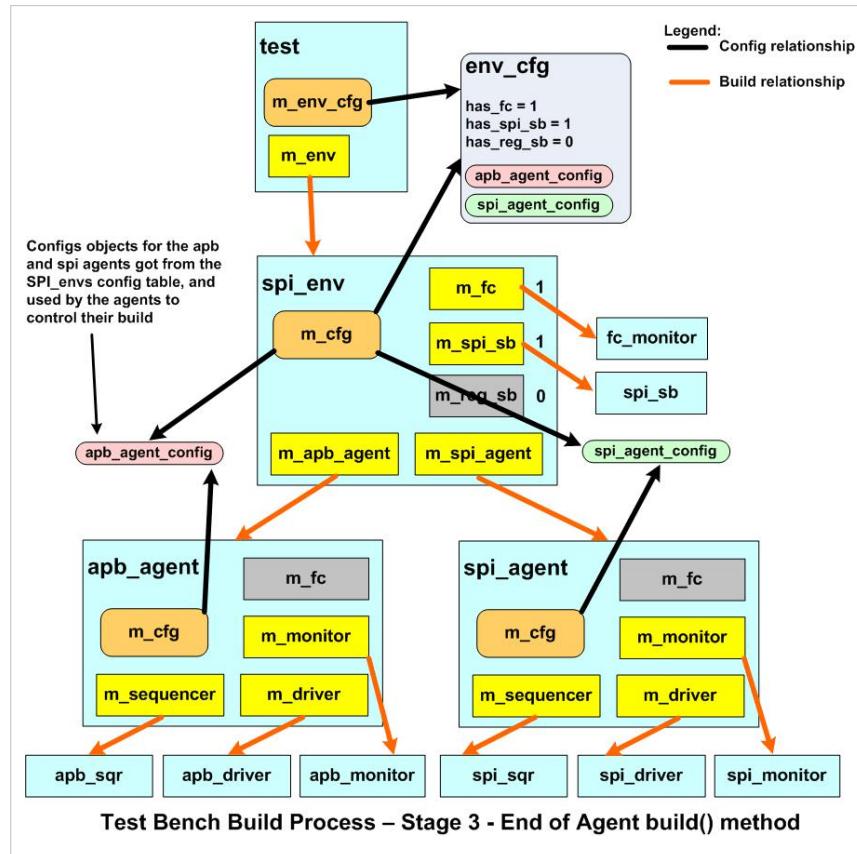


### Coding Convention - Name Argument For Factory Create Method Should Match Local Handle

The create method takes two arguments, one is a name string and the other is a pointer to the parent uvm\_component class object. The values of these arguments are used to create an entry in a linked list which the UVM uses to locate uvm\_components in a pseudo hierarchy, this list is used in the messaging and configuration mechanisms. By convention, the name argument string should be the same as the declaration handle of the component and the parent argument should be the keyword "this" so that it refers to the uvm\_component in which the component is being created. Using the same name as the handle makes it easier to cross reference paths and handles. In the previous code snippet, the spi\_env is created in the test using its declaration handle m\_env. This means that, after the end of the build process, the UVM path to the spi\_env would be "spi\_test.m\_env".

## Hierarchical Build Process

The build phase in the UVM works top down. Once the test class has been constructed, its build method is called, and then the build method of its child(ren) is called. In turn, the build methods of each of the child nodes through the testbench tree are called until the whole hierarchy has been constructed. This deferred approach to construction means that each build method can affect what happens in the build process of components at lower levels in the testbench hierarchy. For instance, if an agent is configured to be passive, the build process for the agent omits the creation of the agents sequencer and driver since these are only required if the agent is active.



## The Hierarchical Connection Process

Once the build phase has completed, the UVM testbench component hierarchy is in place and the individual components have been constructed and linked into the component hierarchy linked list. The connect phase follows the build phase, and works from the bottom of the hierarchy tree to the top. Its purpose is to make connections between TLM classes, assign virtual interface pointers to their handles and to make any other assignments for resources such as register models.

Configuration objects are used during the connection process since they may contain references to virtual interfaces or they may contain information that guides the connection process. For instance, inside an agent, the virtual interface assignment to a driver and the TLM connection between a driver and its sequencer can only be made if the agent is active.

## Examples

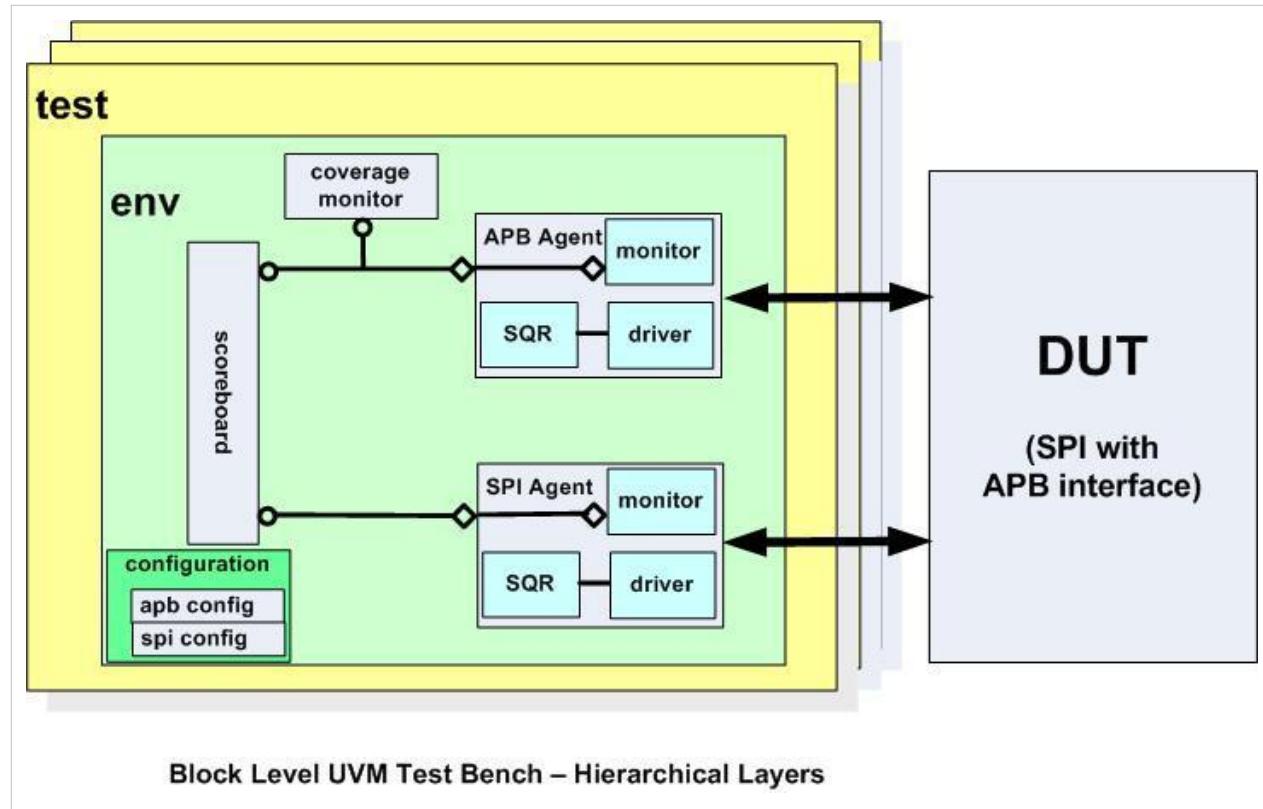
The build process is best illustrated by looking at some examples to illustrate how different types of component hierarchy are built up:

A block level testbench containing an agent

An integration level testbench

# Testbench/Blocklevel

As an example of a block level test bench, we are going to consider a test bench built to verify a SPI Master DUT. In this case, the UVM environment has two agents - an APB agent to handle bus transfers on its APB slave port, and a SPI agent to handle SPI protocol transfers on its SPI port. The structure of the overall UVM verification environment is illustrated in the block diagram. We shall go through each layer of the test bench and describe how it is put together from the top down.



## The Test Bench Module

The top level test bench module is used to encapsulate the SPI Master DUT and connect it to the `apb_if` and `spi_if` SystemVerilog interfaces. There is also an initial block which generates a clock and a reset signal for the APB interface.

In the initial block of the test bench, handles for the APB, SPI and INTR (interrupt) virtual interfaces are put into the UVM top configuration space using `uvm_config_db::set`. Then the `run_test()` method is called - this causes the specified test to be constructed and the processing of the UVM phases to start.

```
module top_tb;

`include "timescale.v"

import uvm_pkg::*;

import spi_test_lib_pkg::*;


```

```
// PCLK and PRESETn
//
logic PCLK;
logic PRESETn;

//
// Instantiate the interfaces:
//
apb_if APB(PCLK, PRESETn); // APB interface
spi_if SPI(); // SPI Interface
intr_if INTR(); // Interrupt

//
// DUT
spi_top DUT(
  // APB Interface:
  .PCLK(PCLK),
  .PRESETN(PRESETn),
  .PSEL(APB.PSEL[0]),
  .PADDR(APB.PADDR[4:0]),
  .PWDATA(APB.PWDATA),
  .PRDATA(APB.PRDATA),
  .PENABLE(APB.PENABLE),
  .PREADY(APB.PREADY),
  .PSLVERR(),
  .PWRITE(APB.PWRITE),
  // Interrupt output
  .IRQ(INTR.IRQ),
  // SPI signals
  .ss_pad_o(SPI.cs),
  .sclk_pad_o(SPI.clk),
  .mosi_pad_o(SPI.mosi),
  .miso_pad_i(SPI.miso)
);

// UVM initial block:
// Virtual interface wrapping & run_test()
initial begin
  uvm_config_db #(virtual apb_if)::set( null , "uvm_test_top" , "APB_vif" , APB);
  uvm_config_db #(virtual spi_if)::set( null , "uvm_test_top" , "SPI_vif" , SPI);
  uvm_config_db #(virtual intr_if)::set( null , "uvm_test_top" , "INTR_vif", INTR);
  run_test();
end
```

```

//  

// Clock and reset initial block:  

//  

initial begin  

  PCLK = 0;  

  PRESETn = 0;  

  repeat(8) begin  

    #10ns PCLK = ~PCLK;  

  end  

  PRESETn = 1;  

  forever begin  

    #10ns PCLK = ~PCLK;  

  end  

end  

endmodule: top_tb

```

## The Test

The next phase in the UVM construction process is the build phase. For the SPI block level example this means building the `spi_env` component, having first created and prepared all of the configuration objects that are going to be used by the environment. The configuration and build process is likely to be common to most test cases, so it is usually good practice to create a test base class that can be extended to create specific tests.

In the SPI example, the configuration object for the `spi_env` contains handles for the SPI and APB configuration objects. This allows the env configuration object to be used to pass all of the configuration objects to the env. The build method in the `spi_env` is then responsible for passing on these sub-configurations. This "Russian Doll" approach to nesting configurations is used since it is scalable for many levels of hierarchy.

Before the configuration objects for the agents are assigned to their handles in the env configuration block, they are constructed, have their virtual interfaces assigned, using the `uvm_config_db::get` method, and then they are configured. The APB agent may well be configured differently between test cases and so its configuration process has been split out into a separate virtual method in the base class. This allows inheriting test classes to overload this method and configure the APB agent differently.

The following code is for the `spi_test_base` class:

```

//  

// Class Description:  

//  

//  

class spi_test_base extends uvm_test;  

// UVM Factory Registration Macro  

//  

`uvm_component_utils(spi_test_base)

```

```
//-----
// Data Members
//-----

//-----
// Component Members
//-----

// The environment class
spi_env m_env;

// Configuration objects
spi_env_config m_env_cfg;
apb_agent_config m_apb_cfg;
spi_agent_config m_spi_cfg;

// Register map
spi_register_map spi_rm;

//-----
// Methods
//-----

extern virtual function void configure_apb_agent(apb_agent_config cfg);
// Standard UVM Methods:
extern function new(string name = "spi_test_base", uvm_component parent = null);
extern function void build_phase( uvm_phase phase );

endclass: spi_test_base

function spi_test_base::new(string name = "spi_test_base", uvm_component parent = null);
  super.new(name, parent);
endfunction

// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // Register map - Keep reg_map a generic name for vertical reuse reasons
  spi_rm = new("reg_map", null);
  m_env_cfg.spi_rm = spi_rm;
  m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
  configure_apb_agent(m_apb_cfg);
  if( !uvm_config_db #(virtual apb_if)::get(this, "", "APB_vif",m_apb_cfg.APB) ) `uvm_error(...)
  m_env_cfg.m_apb_agent_cfg = m_apb_cfg;
  // The SPI is not configured as such
  m_spi_cfg = spi_agent_config::type_id::create("m_spi_cfg");
  if( !uvm_config_db #(virtual spi_if)::get(this, "", "SPI_vif",m_spi_cfg.SPI) ) `uvm_error(...)
```

```

m_spi_cfg.has_functional_coverage = 0;
m_env_cfg.m_spi_agent_cfg = m_spi_cfg;
// Insert the interrupt virtual interface into the env_config:
if( !uvm_config_db #(virtual intr_if)::get(this, "", "INTR_vif",m_env_cfg.INTR) ) `uvm_error(...)

uvm_config_db #( spi_env_config )::set( this , "*", "spi_env_config", m_env_cfg);
m_env = spi_env::type_id::create("m_env", this);
// Override for register adapter:
register_adapter_base::type_id::set_inst_override(apb_register_adapter::get_type(), "spi_bus.adapter");
endfunction: build_phase

//


// Convenience function to configure the apb agent
//


// This can be overloaded by extensions to this base class
function void spi_test_base::configure_apb_agent(apb_agent_config cfg);
  cfg.active = UVM_ACTIVE;
  cfg.has_functional_coverage = 0;
  cfg.has_scoreboard = 0;
  // SPI is on select line 0 for address range 0-18h
  cfg.no_select_lines = 1;
  cfg.start_address[0] = 32'h0;
  cfg.range[0] = 32'h18;
endfunction: configure_apb_agent

```

To create a specific test case, the `spi_test_base` class is extended, and this allows the test writer to take advantage of the configuration and build process defined in the parent class and means that he only needs to add a run method. In the following (simplistic and to be updated) example, the run method instantiates a virtual sequence and starts it on the virtual sequencer in the env. All of the configuration process is carried out by the `super.build()` method call in the build method.

```

//
// Class Description:
//
//
class spi_test extends spi_test_base;

// UVM Factory Registration Macro
//
`uvm_component_utils(spi_test)

//-----
// Methods
//-----

```

```
// Standard UVM Methods:
extern function new(string name = "spi_test", uvm_component parent = null);
extern function void build_phase( uvm_phase phase );
extern task reset_phase( uvm_phase phase );
extern task main_phase( uvm_phase phase );

endclass: spi_test

function spi_test::new(string name = "spi_test", uvm_component parent = null);
  super.new(name, parent);
endfunction

// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void spi_test::build_phase( uvm_phase phase );
  super.build_phase( phase );
endfunction: build_phase

task spi_test::reset_phase( uvm_phase phase );
  check_reset_seq reset_test_seq = check_reset_seq::type_id::create("rest_test_seq");

  phase.raise_objection( this , "Starting reset_test_seq in reset phase" );
  reset_test_seq.start(m_env.m_v_sqr.apb);
  phase.drop_objection( this , "Finished reset_test_seq in reset phase" );
endtask: reset_phase

task spi_test::main_phase( uvm_phase phase );
  send_spi_char_seq spi_char_seq = send_spi_char_seq::type_id::create("spi_char_seq");

  phase.raise_objection( this , "Starting spi_char_seq in main phase" );
  spi_char_seq.start(m_env.m_apb_agent.m_sequencer);
  #100ns;
  phase.drop_objection( this , "Finished spi_char_seq in main phase" );
endtask: main_phase
```

## The env

The next level in the SPI UVM environment is the spi\_env. This class contains a number of sub-components, namely the SPI and APB agents, a scoreboard, a functional coverage monitor and a virtual sequencer. Which of these sub-components gets built is determined by variables in the spi\_env configuration object.

In this case, the spi\_env configuration object also contains a virtual interface and a method for detecting an interrupt. This will be used by *sequences* running on the *virtual sequencer*. The contents of the spi\_env\_config class are as follows:

```
//  
// Class Description:  
//  
//  
class spi_env_config extends uvm_object;  
  
localparam string s_my_config_id = "spi_env_config";  
localparam string s_no_config_id = "no config";  
localparam string s_my_config_type_error_id = "config type error";  
  
// UVM Factory Registration Macro  
//  
`uvm_object_utils(spi_env_config)  
  
// Interrupt Virtual Interface - used in the wait for interrupt task  
//  
virtual intr_if INTR;  
  
//-----  
// Data Members  
//-----  
// Whether env analysis components are used:  
bit has_functional_coverage = 0;  
bit has_spi_functional_coverage = 1;  
bit has_reg_scoreboard = 0;  
bit has_spi_scoreboard = 1;  
// Whether the various agents are used:  
bit has_apb_agent = 1;  
bit has_spi_agent = 1;  
// Configurations for the sub_components  
apb_agent_config m_apb_agent_cfg;  
spi_agent_config m_spi_agent_cfg;  
// SPI Register model  
uvm_register_map spi_rm;  
  
//-----
```

```

// Methods
//-----
extern task wait_for_interrupt;
extern function bit is_interrupt_cleared;
// Standard UVM Methods:
extern function new(string name = "spi_env_config");

endclass: spi_env_config

function spi_env_config::new(string name = "spi_env_config");
  super.new(name);
endfunction

// This task is a convenience method for sequences waiting for the interrupt
// signal
task spi_env_config::wait_for_interrupt;
  @(posedge INTR.IRQ);
endtask: wait_for_interrupt

// Check that interrupt has cleared:
function bit spi_env_config::is_interrupt_cleared;
  if(INTR.IRQ == 0)
    return 1;
  else
    return 0;
endfunction: is_interrupt_cleared

```

In this example, there are build configuration field bits for each sub-component. This gives the env the ultimate flexibility for reuse.

During the spi\_env's build phase, a handle to the spi\_env\_config is retrieved from the configuration space using uvm\_config\_db get(). Then the build process tests the various has\_<sub\_component> fields in the configuration object to determine whether to build a sub-component. In the case of the APB and SPI agents, there is an additional step which is to unpack the configuration objects for each of the agents from the envs configuration object and then to set the agent configuration objects in the envs configuration table after any local modification.

In the connect phase, the spi\_env configuration object is again used to determine which TLM connections to make.

```

// 
// Class Description:
// 
// 
class spi_env extends uvm_env;

// UVM Factory Registration Macro
// 
`uvm_component_utils(spi_env)

```

```
//-----
// Data Members
//-----
apb_agent m_apb_agent;
spi_agent m_spi_agent;
spi_env_config m_cfg;
spi_register_coverage m_reg_cov_monitor;
spi_reg_functional_coverage m_func_cov_monitor;
spi_scoreboard m_scoreboard;
//-----
// Constraints
//-----

//-----
// Methods
//-----

// Standard UVM Methods:
extern function new(string name = "spi_env", uvm_component parent = null);
extern function void build_phase( uvm_phase phase );
extern function void connect_phase( uvm_phase phase );

endclass:spi_env

function spi_env::new(string name = "spi_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

function void spi_env::build_phase( uvm_phase phase );
  if(!uvm_config_db #( spi_env_config )::get( this , "", "spi_env_config" , m_cfg ) begin
    `uvm_error("build_phase", "unable to get spi_env_config")
  end
  if(m_cfg.has_apb_agent) begin
    uvm_config_db #( apb_agent_config )::set( this , "m_apb_agent*", "apb_agent_config", m_cfg.m_apb_agent_cfg );
    m_apb_agent = apb_agent::type_id::create("m_apb_agent", this);
  end
  if(m_cfg.has_spi_agent) begin
    uvm_config_db #( spi_agent_config )::set("m_spi_agent*", "spi_agent_config", m_cfg.m_spi_agent_cfg );
    m_spi_agent = spi_agent::type_id::create("m_spi_agent", this);
  end
  if(m_cfg.has_functional_coverage) begin
    m_reg_cov_monitor = spi_register_coverage::type_id::create("m_reg_cov_monitor", this);
  end
  if(m_cfg.has_spi_functional_coverage) begin
```

```
m_func_cov_monitor = spi_reg_functional_coverage::type_id::create("m_func_cov_monitor", this);
end

if(m_cfg.has_spi_scoreboard) begin
  m_scoreboard = spi_scoreboard::type_id::create("m_scoreboard", this);
end

endfunction:build_phase

function void spi_env::connect_phase( uvm_phase phase );
  if(m_cfg.has_functional_coverage) begin
    m_apb_agent.ap.connect(m_reg_cov_monitor.analysis_export);
  end

  if(m_cfg.has_spi_functional_coverage) begin
    m_apb_agent.ap.connect(m_func_cov_monitor.analysis_export);
  end

  if(m_cfg.has_spi_scoreboard) begin
    m_apb_agent.ap.connect(m_scoreboard.apb.analysis_export);
    m_spi_agent.ap.connect(m_scoreboard.spi.analysis_export);
    m_scoreboard.spi_rm = m_cfg.spi_rm;
  end
end

endfunction: connect_phase
```

## The Agents

Since the UVM build process is top down, the SPI and APB agents are constructed next. The article on the *agent build process* describes how the APB agent is configured and built, the SPI agent follows the same process.

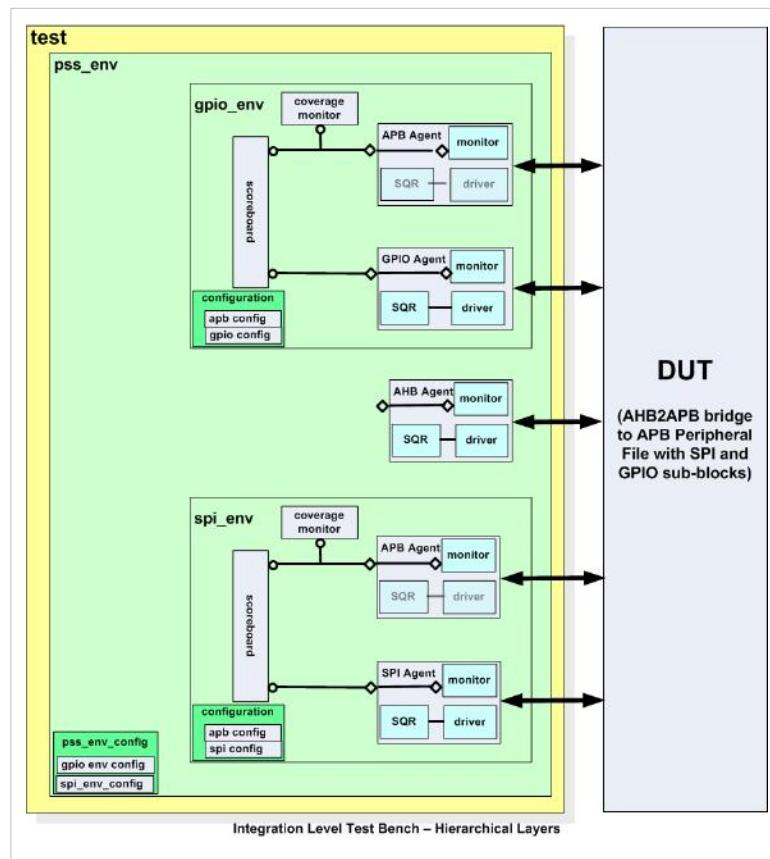
The components within the agents are at the bottom of the test bench hierarchy, so the build process terminates there.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Testbench/IntegrationLevel

This test bench example is one that takes two block level verification environments and shows how they can be reused at a higher level of integration. The principles that are illustrated in the example are applicable to repeated rounds of vertical reuse.

The example takes the SPI block level example and integrates it with another block level verification environment for a GPIO DUT. The hardware for the two blocks has been integrated into a Peripheral Sub-System (PSS) which uses an AHB to APB bus bridge to interface with the APB interfaces on the SPI and GPIO blocks. The environments from the block level are encapsulated by the pss\_env, which also includes an AHB agent to drive the exposed AHB bus interface. In this configuration, the block level APB bus interfaces are no longer exposed, and so the APB agents are put into passive mode to monitor the APB traffic. The stimulus needs to drive the AHB interface and register layering enables reuse of block level stimulus at the integration level.



We shall now go through the test bench and the build process from the top down, starting with the top level test bench module.

## Top Level Test Bench Module

As with the block level test bench example, the top level test bench module is used to encapsulate the DUT, connect interfaces to the DUT signal pins and then set the virtual interface containers up in the configuration space before calling `run_test()`. The main differences between this code and the block level test bench code are that there are more interfaces and that there is a need to bind to some internal signals to monitor the APB bus. The DUT is wrapped by a module which connects its I/O signals to the interfaces used in the UVM test bench. The internal signals are bound to the APB interface

using the binder module:

```
module top_tb;

import uvm_pkg::*;
import pss_test_lib_pkg::*;

// PCLK and PRESETn
//
logic HCLK;
logic HRESETn;

//
// Instantiate the interfaces:
//
apb_if APB(HCLK, HRESETn); // APB interface - shared between passive agents
ahb_if AHB(HCLK, HRESETn); // AHB interface
spi_if SPI(); // SPI Interface
intr_if INTR(); // Interrupt
gpio_if GPO();
gpio_if GPI();
gpio_if GPOE();
icpit_if ICPIT();
serial_if UART_RX();
serial_if UART_TX();
modem_if MODEM();

// Binder
binder probe();

// DUT Wrapper:
pss_wrapper wrapper(.ahb(AHB),
    .spi(SPI),
    .gpi(GPI),
    .gpo(GPO),
    .gpoe(GPOE),
    .icpit(ICPIT),
    .uart_rx(UART_RX),
    .uart_tx(UART_TX),
    .modem(MODEM));

//
// UVM initial block:
// Virtual interface wrapping & run_test()
initial begin
```

```
uvm_config_db #(virtual apb_if)::set( null , "uvm_test_top" , "APB_vif" , APB);
uvm_config_db #(virtual ahb_if)::set( null , "uvm_test_top" , "AHB_vif" , AHB);
uvm_config_db #(virtual spi_if)::set( null , "uvm_test_top" , "SPI_vif" , SPI);
uvm_config_db #(virtual intr_if)::set( null , "uvm_test_top" , "INTR_vif" , INTR);
uvm_config_db #(virtual gpio_if)::set( null , "uvm_test_top" , "GPO_vif" , GPO);
uvm_config_db #(virtual gpio_if)::set( null , "uvm_test_top" , "GPOE_vif" , GPOE);
uvm_config_db #(virtual gpio_if)::set( null , "uvm_test_top" , "GPI_vif" , GPI);
uvm_config_db #(virtual icpit_if)::set( null , "uvm_test_top" , "ICPIT_vif" , ICPIT);
uvm_config_db #(virtual serial_if)::set( null , "uvm_test_top" , "UART_RX_vif" , UART_RX);
uvm_config_db #(virtual serial_if)::set( null , "uvm_test_top" , "UART_TX_vif" , UART_TX);
uvm_config_db #(virtual modem_if)::set( null , "uvm_test_top" , "MODEM_vif" , MODEM);
run_test();
end

//  

// Clock and reset initial block:  

//  

initial begin
    HCLK = 0;
    HRESETn = 0;
    repeat(8) begin
        #10ns HCLK = ~HCLK;
    end
    HRESETn = 1;
    forever begin
        #10ns HCLK = ~HCLK;
    end
end

// Clock assignments:
assign GPO.clk = HCLK;
assign GPOE.clk = HCLK;
assign GPI.clk = HCLK;
assign ICPIT.PCLK = HCLK;

endmodule: top_tb
```

## The Test

Like the block level test, the integration level test should have the common build and configuration process captured in a base class that subsequent test cases can inherit from. As can be seen from the example, there is more configuration to do and so the need becomes more compelling.

The configuration object for the pss\_env contains handles for the configuration objects for the spi\_env and the gpio\_env. In turn, the sub-env configuration objects contain handles for their agent sub-component configuration objects. The pss\_env is responsible for unnesting the spi\_env and gpio\_env configuration objects and setting them in its configuration table, making any local changes necessary. In turn, the spi\_env and the gpio\_env put their agent configurations into their configuration table.

The pss test base class is as follows:

```

//  

// Class Description:  

//  

//  

//  

class pss_test_base extends uvm_test;  

// UVM Factory Registration Macro  

//  

`uvm_component_utils(pss_test_base)  

//-----  

// Data Members  

//-----  

//-----  

// Component Members  

//-----  

// The environment class  

pss_env m_env;  

// Configuration objects  

pss_env_config m_env_cfg;  

spi_env_config m_spi_env_cfg;  

gpio_env_config m_gpio_env_cfg;  

//uart_env_config m_uart_env_cfg;  

apb_agent_config m_spi_apb_agent_cfg;  

apb_agent_config m_gpio_apb_agent_cfg;  

ahb_agent_config m_ahb_agent_cfg;  

spi_agent_config m_spi_agent_cfg;  

gpio_agent_config m_GPO_agent_cfg;  

gpio_agent_config m_GPI_agent_cfg;  

gpio_agent_config m_GPOE_agent_cfg;  

// Register map

```

```

pss_register_map pss_rm;

//-----
// Methods
//-----

// Standard UVM Methods:

extern function new(string name = "spi_test_base", uvm_component parent = null);
extern function void build_phase( uvm_phase phase);
extern virtual function void configure_apb_agent(apb_agent_config cfg, int index, logic[31:0] start_address, logic[31:0] range);

extern task run_phase( uvm_phase phase );

endclass: pss_test_base

function pss_test_base::new(string name = "spi_test_base", uvm_component parent = null);
  super.new(name, parent);
endfunction

// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces

function void pss_test_base::build_phase( uvm_phase );
  m_env_cfg = pss_env_config::type_id::create("m_env_cfg");
  // Register map - Keep reg_map a generic name for vertical reuse reasons
  pss_rm = new("reg_map", null);
  m_env_cfg.pss_rm = pss_rm;

  // SPI Sub-env configuration:
  m_spi_env_cfg = spi_env_config::type_id::create("m_spi_env_cfg");
  m_spi_env_cfg.spi_rm = pss_rm;
  // apb agent in the SPI env:
  m_spi_env_cfg.has_apb_agent = 1;
  m_spi_apb_agent_cfg = apb_agent_config::type_id::create("m_spi_apb_agent_cfg");
  configure_apb_agent(m_spi_apb_agent_cfg, 0, 32'h0, 32'h18);
  if( !uvm_config_db #(virtual apb_if)::get(this, "", "APB_vif",m_spi_apb_agent_cfg.APB) ) `uvm_error(...)

  m_spi_env_cfg.m_apb_agent_cfg = m_spi_apb_agent_cfg;
  // SPI agent:
  m_spi_agent_cfg = spi_agent_config::type_id::create("m_spi_agent_cfg");
  if( !uvm_config_db #(virtual spi_if)::get(this, "", "SPI_vif",m_spi_agent_cfg.SPI) ) `uvm_error(...)

  m_spi_env_cfg.m_spi_agent_cfg = m_spi_agent_cfg;
  m_env_cfg.m_spi_env_cfg = m_spi_env_cfg;
  if( !uvm_config_db #( spi_env_config )::set("*, "", "spi_env_config", m_spi_env_cfg) ) `uvm_error(...)

  // GPIO env configuration:
  m_gpio_env_cfg = gpio_env_config::type_id::create("m_gpio_env_cfg");
  m_gpio_env_cfg gpio_rm = pss_rm;
  m_gpio_env_cfg.has_apb_agent = 1; // APB agent used

```

```

m_gpio_apb_agent_cfg = apb_agent_config::type_id::create("m_gpio_apb_agent_cfg");
configure_apb_agent(m_gpio_apb_agent_cfg, 1, 32'h100, 32'h124);
if( !uvm_config_db #(virtual apb_if)::get(this, "", "APB_vif",m_gpio_apb_agent_cfg.APB) ) `uvm_error(...)

m_gpio_env_cfg.m_apb_agent_cfg = m_gpio_apb_agent_cfg;
m_gpio_env_cfg.has_functional_coverage = 1; // Register coverage no longer valid

// GPO agent

m_GPO_agent_cfg = gpio_agent_config::type_id::create("m_GPO_agent_cfg");
uvm_config_db #(virtual gpio_if)::get(this, "GPO_vif",m_GPO_agent_cfg.GPIO);
m_GPO_agent_cfg.active = UVM_PASSIVE; // Only monitors
m_gpio_env_cfg.m_GPO_agent_cfg = m_GPO_agent_cfg;
m_GPOE_agent_cfg = gpio_agent_config::type_id::create("m_GPOE_agent_cfg");
uvm_config_db #(virtual gpio_if)::get(this, "GPOE_vif",m_GPOE_agent_cfg.GPIO);
m_GPOE_agent_cfg.active = UVM_PASSIVE; // Only monitors
m_gpio_env_cfg.m_GPOE_agent_cfg = m_GPOE_agent_cfg;
m_GPI_agent_cfg = gpio_agent_config::type_id::create("m_GPI_agent_cfg");
if( !uvm_config_db #(virtual gpio_if)::get(this, "", "GPI_vif",m_GPI_agent_cfg.GPIO) ) `uvm_error(...)

m_gpio_env_cfg.m_GPI_agent_cfg = m_GPI_agent_cfg;
// GPIO Aux agent not present

m_gpio_env_cfg.has_AUX_agent = 0;
m_gpio_env_cfg.has_functional_coverage = 1;
m_gpio_env_cfg.has_reg_scoreboard = 0;
m_gpio_env_cfg.has_out_scoreboard = 1;
m_gpio_env_cfg.has_in_scoreboard = 1;
m_env_cfg.m_gpio_env_cfg = m_gpio_env_cfg;
uvm_config_db #( gpio_env_config )::set( this, "", "gpio_env_config", m_gpio_env_cfg );
// AHB Agent

m_ahb_agent_cfg = ahb_agent_config::type_id::create("m_ahb_agent_cfg");
if( !uvm_config_db #(virtual ahb_if)::get(this, "", "AHB_vif",m_ahb_agent_cfg.AHB) ) `uvm_error(...)

m_env_cfg.m_ahb_agent_cfg = m_ahb_agent_cfg;
// Add in interrupt line

if( !uvm_config_db #(virtual icpit_if)::get(this, "", "ICPIT_vif",m_env_cfg.ICPIT) ) `uvm_error(...)

uvm_config_db::set( this, "", "pss_env_config", m_env_cfg );
m_env = pss_env::type_id::create("m_env", this);
// override for register adapters:
register_adapter_base::type_id::set_inst_override(ahb_register_adapter::get_type(), "spi_bus.adapter");
register_adapter_base::type_id::set_inst_override(ahb_register_adapter::get_type(), "gpio_bus.adapter");
endfunction: build_phase

// Convenience function to configure the apb agent
// This can be overloaded by extensions to this base class

```

```

function void pss_test_base::configure_apb_agent(apb_agent_config cfg, int index, logic[31:0] start_address, logic[31:0] range);
    cfg.active = UVM_PASSIVE;
    cfg.has_functional_coverage = 0;
    cfg.has_scoreboard = 0;
    cfg.no_select_lines = 1;
    cfg.apb_index = index;
    cfg.start_address[0] = start_address;
    cfg.range[0] = range;
endfunction: configure_apb_agent

task pss_test_base::run_phase( uvm_phase phase );
endtask: run_phase

```

Again, a test case that extends this base class would populate its run method to define a virtual sequence that would be run on the virtual sequencer in the env. If there is non-default configuration to be done, then this could be done by populating or overloading the build method or any of the configuration methods.

```

// 
// Class Description:
// 
// 
class pss_test extends pss_test_base;

// UVM Factory Registration Macro
//
`uvm_component_utils(pss_test)

//-----
// Methods
//-----

// Standard UVM Methods:
extern function new(string name = "pss_test", uvm_component parent = null);
extern function void build_phase( ovm_phase phase );
extern task run_phase( uvm_phase phase );

endclass: pss_test

function pss_test::new(string name = "pss_test", uvm_component parent = null);
    super.new(name, parent);
endfunction

// Build the env, create the env configuration

```

```

// including any sub configurations and assigning virtual interfaces
function void pss_test::build_phase( uvm_phase phase);
    super.build_phase( uvm_phase phase );
endfunction: build_phase

// Create, initialise and then run the virtual sequence
task pss_test::run_phase( uvm_phase phase );
    bridge_basic_rw_vseq t_seq = bridge_basic_rw_vseq::type_id::create("t_seq");

    phase.raise_objection( this , "Starting PSS test");

    init_vseq(t_seq);

    repeat(10) begin
        t_seq.start(null);
    end

    phase.drop_objection( this , "Finished PSS test");
endtask: run_phase

```

## The PSS env

The PSS env build process retrieves the configuration object and constructs the various sub-envs, after testing the various has\_<sub-component> fields in order to determine whether the env is required by the test case. If the sub-env is to be present, the sub-envs configuration object is set in the PSS envs configuration table. The connect method is used to make connections between TLM ports and exports between monitors and analysis components such as scoreboards.

```

// 
// Class Description:
// 
// 
class pss_env extends uvm_env;

// UVM Factory Registration Macro
// 
`uvm_component_utils(pss_env)

//-----
// Data Members
//-----
pss_env_config m_cfg;
//-----
// Sub Components
//-----
spi_env m_spi_env;

```

```
gpio_env m_gpio_env;
ahb_agent m_ahb_agent;
pss_virtual_sequencer m_vsqr;
//-----
// Methods
//-----

// Standard UVM Methods:

extern function new(string name = "pss_env", uvm_component parent = null);
// Only required if you have sub-components
extern function void build_phase( uvm_phase phase );
// Only required if you have sub-components which are connected
extern function void connect_phase( uvm_phase phase );

endclass: pss_env

function pss_env::new(string name = "pss_env", uvm_component parent = null);
  super.new(name, parent);
endfunction

// Only required if you have sub-components
function void pss_env::build_phase( uvm_phase phase );
  if( !uvm_config_db #( pss_env_config )::get_config(this,"","pss_env_config",m_cfg) ) `uvm_error(...)

  if(m_cfg.has_spi_env) begin
    uvm_config_db::set( this , "m_spi_env*", "spi_env_config", m_cfg.m_spi_env_cfg);
    m_spi_env = spi_env::type_id::create("m_spi_env", this);
  end

  if(m_cfg.has_gpio_env) begin
    uvm_config_db #( gpio_env_config )::set("m_gpio_env*", "gpio_env_config", m_cfg.m_gpio_env_cfg);
    m_gpio_env = gpio_env::type_id::create("m_gpio_env", this);
  end

  if(m_cfg.has_ahb_agent) begin
    uvm_config_db #( ahb_agent_config )::set( this , "m_ahb_agent*", "ahb_agent_config", m_cfg.m_ahb_agent_cfg);
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent", this);
  end
endfunction: build_phase

// Only required if you have sub-components which are connected
function void pss_env::connect_phase( uvm_phase phase );
  // Inter-component TLM connections
endfunction: connect_phase
```

## The rest of the test bench hierarchy

The build process continues top-down with the sub-envs being conditionally constructed as illustrated in the block level test bench example and the agents contained within the sub-envs being built as described in the agent example.

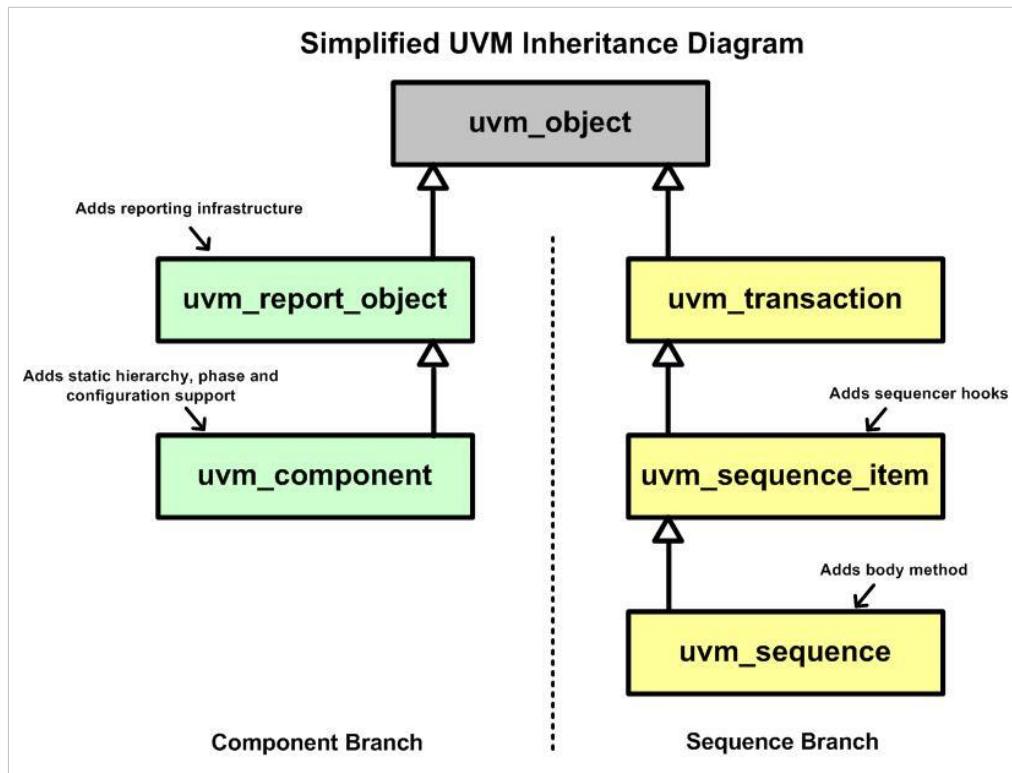
## Further levels of integration

Vertical reuse for further levels of integration can be achieved by extending the process described for the PSS example. Each level of integration adds another layer, so for instance a level 2 integration environment would contain two or more level 1 envs and the level 2 env configuration object would contain nested handles for the level 1 env configuration objects. Obviously, at the test level of the hierarchy the amount of code increases for each round of vertical reuse, but further down the hierarchy, the configuration and build process has already been implemented in the previous generation of vertical layering.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Component

An UVM testbench is built from component objects extended from the `uvm_component` base class. When an `uvm_component` object is created, it becomes part of the testbench hierarchy which remains in place for the duration of the simulation. This contrasts with the sequence branch of the `uvm` class hierarchy, where objects are transient - they are created, used and then garbage collected when dereferenced.



The `uvm_component` static hierarchy is used by the reporting infrastructure for printing out the scope of the component creating a report message, by the configuration process to determine which components can access a configuration object, and by the factory for factory overrides. This static hierarchy is represented by a linked list which is built up as each component is created, the components location in the hierarchy is determined by the name and parent arguments passed to its create method.

For instance, in the code fragment below, an `apb_agent` component is being created within the `spi_env`, which in turn is created inside a test as `m_env`. The hierarchical path to the agent will be `"uvm_test_top.m_env.m_apb_agent"` and any references to it would need to use this string.

```

//  

// Hierarchical name example  

//  

class spi_env extends uvm_env;  

//....  

apb_agent m_apb_agent; // Declaration of the apb agent handle

```

```

// ...

function void build_phase(uvm_phase phase);

  // Create the apb_agent:
  //

  // Name string argument is the same as the handle name
  // The parent argument is 'this' - i.e. the spi_env
  //

  // The spi_env has a hierarchical path string "top.m_env" this is concatenated
  // with the name string to arrive at "uvm_test_top.m_env.m_apb_agent" as the
  // hierarchical reference string for the apb_agent

  m_apb_agent = apb_agent::type_id::create("m_apb_agent", this);

// ...

endfunction: build_phase

// ....
endclass: spi_env

```

The `uvm_component` class inherits from the `uvm_report_object` which contains the functionality required to support the UVM Messaging infrastructure. The reporting process uses the component static hierarchy to add the scope of a component to the report message string.

The `uvm_component` base class template has a virtual method for each of the UVM Phases and these are populated as required, if a phase level virtual method is not implemented then the component does not participate in that phase.

Also built into the `uvm_component` base class is support for a configuration table which is used to store configuration objects which are relevant to a components child nodes in the testbench hierarchy. When the `config_db` API is used, this static hierarchy is used as part of the path mechanism to control which components are able to access a configuration object.

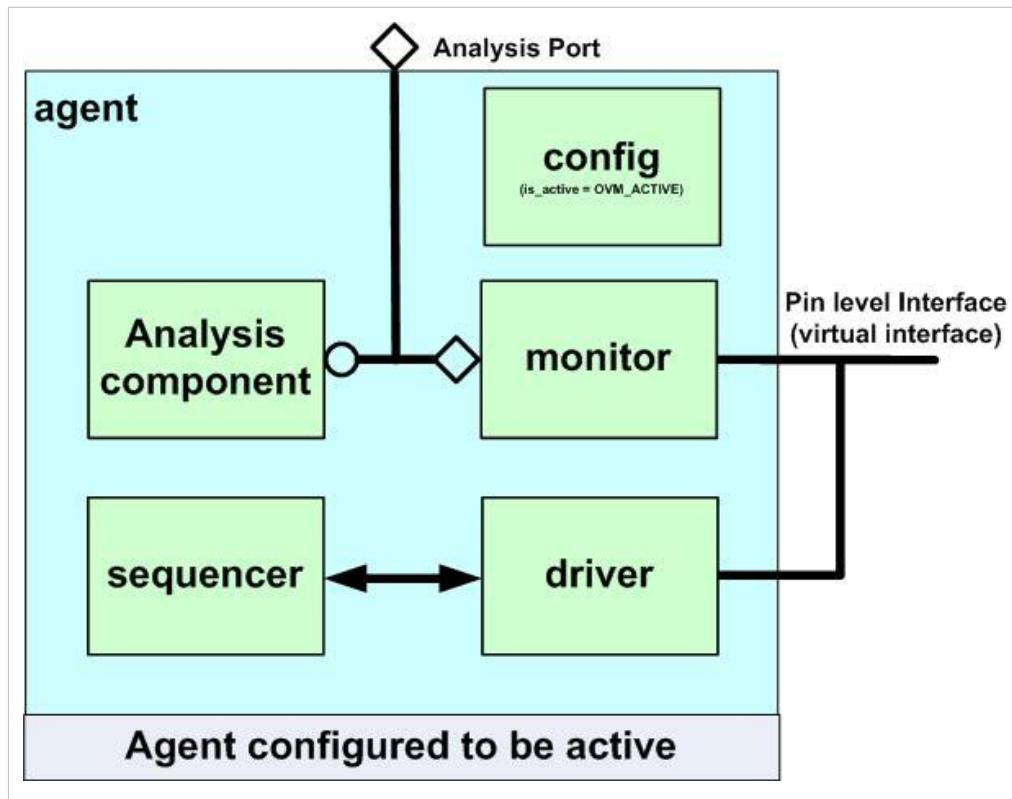
In order to provide flexibility in configuration and to allow the UVM testbench hierarchy to be built in an intelligent way, `uvm_components` are registered with the UVM factory. When an UVM component is created during the build phase, the factory is used to construct the component object. Using the factory allows a component to be swapped for one of a derived type using a factory override. This can be a useful technique for changing the functionality of a testbench without having to recompile. There are a number of coding conventions that are required for the implementation to work and these are outlined in the article on the UVM Factory.

The UVM package contains a number of extensions to the `uvm_component` for common testbench components. Most of these extensions are very thin, i.e. they are literally just an extension of the `uvm_component` with a new name space, this means that an `uvm_component` could be used in their stead. However, they can help with self-documentation since they indicate what type of component the class represents. There are also analysis tools available which use these base classes as clues to help them build up a picture of the testbench hierarchy. A number of the extended components instantiate sub-components and are added value building blocks. The following table summarizes the available `uvm_component` derived classes.

Class	Description	Contains sub-components?
uvm_driver	Adds sequence communication sub-components, used with the uvm_sequencer	Yes
uvm_sequencer	Adds sequence communication sub-components, used with the uvm_driver	Yes
uvm_subscriber	A wrapper around an uvm_analysis_export	Yes
uvm_env	Container for the verification components surrounding a DUT, or other envs surrounding a (sub)system	No
uvm_test	Used for the top level test class	No
uvm_monitor	Used for monitors	No
uvm_scoreboard	Used for scoreboards	No
uvm_agent	Used as the agent container class	No

# Agent

A UVM agent can be thought of as a verification component kit for a specific logical interface. The agent is developed as a package that includes a SystemVerilog interface for connecting to the signal pins of a DUT, and a SystemVerilog package that includes the classes that make up the overall agent component. The agent class itself is a top level container class for a driver, a sequencer and a monitor, plus any other verification components such as functional coverage monitors or scoreboards. The agent also has an analysis port which is connected to the analysis port on the monitor, making it possible for a user to connect external analysis components to the agent without having to know how the agent has been implemented. The agent is the lowest level hierarchical block in a testbench and its exact structure is dependent on its configuration which can be varied from one test to another via the agent configuration object.



In order to illustrate the design and implementation of an agent we shall take an APB bus agent and show how it is packaged, configured, built and connected. The APB agent uses an interface which is called `apb_if` and will be stored in a file called `apb_if.sv`. The various class template files for the agent are collected together in a SystemVerilog package which is saved in a file called `apb_agent_pkg.sv`. Any component, such as an env, that uses files from this package will import this package.

```

package apb_agent_pkg;

import uvm_pkg::*;
`include "uvm_macros.svh"

`include "apb_seq_item.svh"
`include "apb_agent_config.svh"
  
```

```

`include "apb_driver.svh"
`include "apb_coverage_monitor.svh"
`include "apb_monitor.svh"
`include "apb_sequencer.svh"
`include "apb_agent.svh"

// Utility Sequences
`include "apb_seq.svh"

endpackage: apb_agent_pkg

```

## The Agent Configuration Object

The agent has a configuration object which is used to define:

- Which of the agent's sub-components get constructed (Topology)
- The handle for the virtual interface used by the driver and the monitor
- The behavior of the agent

By convention, UVM agents have a variable of type UVM\_ACTIVE\_PASSIVE\_e which defines whether the agent is active (UVM\_ACTIVE) with the sequencer and the driver constructed, or passive (UVM\_PASSIVE) with neither the driver or the sequencer constructed. This parameter is called active and by default it is set to UVM\_ACTIVE.

Whether other sub-components are built or not is controlled by other variables which should have descriptive names. For instance, if there is a functional coverage monitor, then the bit that controls whether the functional coverage monitor gets built or not might be called has\_functional\_coverage.

The configuration object will contain a handle for the virtual interface that is used by the driver and the monitor. The configuration object is constructed and configured in the test and it is in the test that the virtual interface handle is assigned to the virtual interface passed in from the testbench module.

The configuration object may also contain other variables which affect the way in which the agent behaves or is configured. For instance, in the configuration object for the apb agent, there are variables which set up the memory map and determine which apb PSEL lines are activated for which addresses.

The configuration class should have all its variables set to common default values.

The following code example shows the configuration object for the apb agent.

```

// 
// Class Description:
// 
// 
class apb_agent_config extends uvm_object;

// UVM Factory Registration Macro
// 
`uvm_object_utils(apb_agent_config)

// Virtual Interface
virtual apb_if APB;

```

```
//-----
// Data Members
//-----
// Is the agent active or passive
uvm_active_passive_enum active = UVM_ACTIVE;
// Include the APB functional coverage monitor
bit has_functional_coverage = 0;
// Include the APB RAM based scoreboard
bit has_scoreboard = 0;
//
// Address decode for the select lines:
int no_select_lines = 1;
logic[31:0] start_address[15:0];
logic[31:0] range[15:0];

//-----
// Methods
//-----

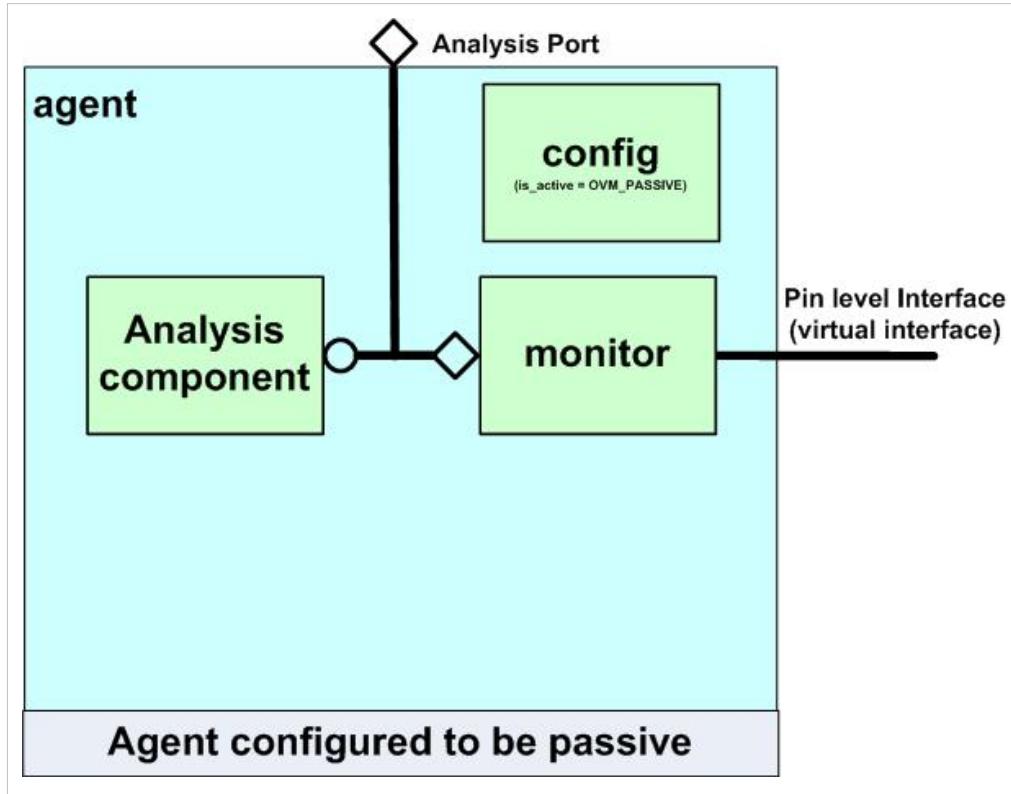
// Standard UVM Methods:
extern function new(string name = "apb_agent_config");

endclass: apb_agent_config

function apb_agent_config::new(string name = "apb_agent_config");
  super.new(name);
endfunction
```

## The Agent Build Phase

The actions that take place during the agent build phase are determined by the contents of its configuration object. The first action during the method is to get a reference to the configuration object. Then, at the points where sub-components are to be constructed, the appropriate configuration field is tested to determine whether they should be constructed or not.



The exceptions to this rule is the monitor which is always built, since it is used whether the agent is active or passive.

## The Agent Connect Phase

Once the agent's sub-components have been constructed, they need to be connected. The usual connections required are:

- Monitor analysis port to the agent's analysis port <sup>1</sup>
- The sequencer's seq\_item\_pull\_export to the driver's seq\_item\_pull\_port (If the agent is active).
- Any agent analysis sub-components analysis exports to the monitor's analysis port
- Assignment of the virtual interfaces in the driver and the monitor to the virtual interface in the configuration object <sup>2</sup>

### Notes:

1. The agent's analysis port handle can be assigned a pointer from the monitor's analysis port. This saves having to construct a separate analysis port object in the agent.
2. Assigning the driver and monitor's virtual interfaces in the agent removes the need for these sub-components to have the overhead of a configuration table lookup.

The following code for the apb agent illustrates how the configuration object determines what happens during the build and connect phases:

```
//  
// Class Description:  
//  
//  
class apb_agent extends uvm_component,  
  
  // UVM Factory Registration Macro  
  //  
  `uvm_component_utils(apb_agent)  
  
//-----  
// Data Members  
//-----  
apb_agent_config m_cfg;  
//-----  
// Component Members  
//-----  
uvm_analysis_port #(apb_seq_item) ap;  
apb_monitor m_monitor;  
apb_sequencer m_sequencer;  
apb_driver m_driver;  
apb_coverage_monitor m_fcov_monitor;  
//-----  
// Methods  
//-----  
  
// Standard UVM Methods:  
extern function new(string name = "apb_agent", uvm_component parent = null);  
extern function void build_phase( uvm_phase phase );  
extern function void connect_phase( uvm_phase phase );  
  
endclass: apb_agent  
  
  
function apb_agent::new(string name = "apb_agent", uvm_component parent = null);  
  super.new(name, parent);  
endfunction  
  
function void apb_agent::build_phase( uvm_phase phase );  
  if( !uvm_config_db #( apb_agent_config )::get(this,"apb_agent_config",m_cfg) ) `uvm_error(...)  
  // Monitor is always present  
  m_monitor = apb_monitor::type_id::create("m_monitor", this);  
  // Only build the driver and sequencer if active  
  if(m_cfg.active == UVM_ACTIVE) begin
```

```
m_driver = apb_driver::type_id::create("m_driver", this);
m_sequencer = apb_sequencer::type_id::create("m_sequencer", this);
end
if(m_cfg.has_functional_coverage) begin
  m_fcov_monitor = apb_coverage_monitor::type_id::create("m_fcov_monitor", this);
end
endfunction: build_phase

function void apb_agent::connect_phase( uvm_phase phsae );
  m_monitor.APB = m_cfg.APB;
  ap = m_monitor.ap;
  // Only connect the driver and the sequencer if active
  if(m_cfg.active == UVM_ACTIVE) begin
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
    m_driver.APB = m_cfg.APB;
  end
  if(m_cfg.has_functional_coverage) begin
    m_monitor.ap.connect(m_fcov_monitor.analysis_export);
  end
endfunction: connect_phase
```

The build process for the APB agent can be followed in the block level testbench example:

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Phasing

---

## Introduction

Phasing is a major area of new functionality in the UVM. Accelera has introduced a number of new technologies:

1. An enhanced set of standard time consuming phases
2. The ability to jump backwards and forwards between phases
3. The ability to add user defined phases and to specify the relationships between them
4. The ability to define domains with different schedules

There are a number of articles in this section which describe our recommendations for using this technology. These include:

1. How to gradually migrate your OVM code to UVM so that it becomes phase aware

The Accellera UVM committee is still developing the use models and APIs for new UVM phasing as it relates to sequences and transactors. In the mean time, our recommendation is to wait until that work is done and the API is stable. There are a number of future articles in this section which will be available here at that time, and which will describe our recommendations for using this technology. These include:

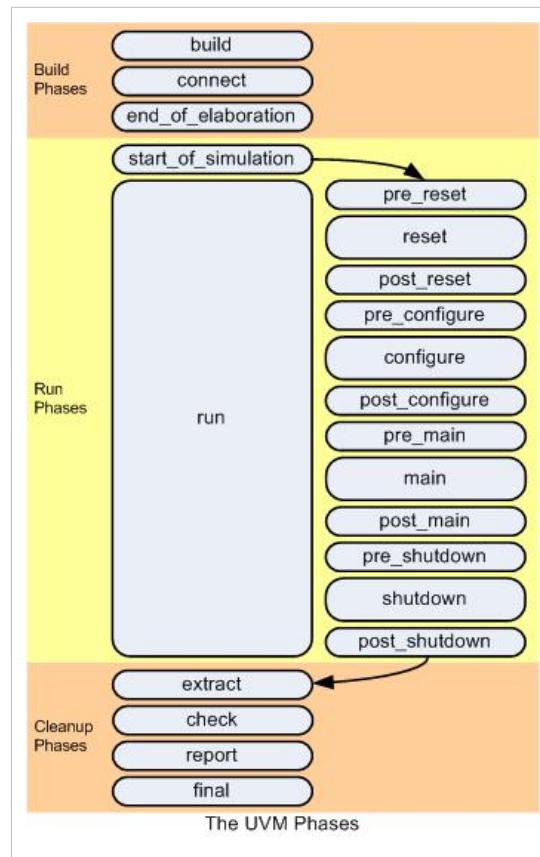
1. How to make your testbench phase aware [Not yet available]
2. How to manage sequences in the context of phasing [Not yet available]
3. How to design reusable transactors that work in the context of phasing [Not yet available]

## The Standard UVM Phases

In order to have a consistent testbench execution flow, the UVM uses phases to order the major steps that take place during simulation. There are three groups of phases, which are executed in the following order:

1. Build phases - where the testbench is configured and constructed
2. Run-time phases - where time is consumed in running the testcase on the testbench
3. Clean up phases - where the results of the testcase are collected and reported

The different phase groups are illustrated in the diagram below. The uvm\_component base class contains virtual methods which are called by each of the different phase methods and these are populated by the testbench component creator according to which phases the component participates in. Using the defined phases allows verification components to be developed in isolation, but still be interoperable since there is a common understanding of what should happen in each phase.



## Starting UVM Phase Execution

To start a UVM testbench, the `run_test()` method has to be called from the static part of the testbench. It is usually called from within an initial block in the top level module of the testbench.

Calling `run_test()` constructs the UVM environment root component and then initiates the UVM phasing. The `run_test()` method can be passed a string argument to define the default type name of an `uvm_component` derived class which is used as the root node of the testbench hierarchy. However, the `run_test()` method checks for a command line plusarg called `UVM_TESTNAME` and uses that plusarg string to lookup a factory registered `uvm_component`, overriding any default type name. By convention, the root node will be derived from a `uvm_test` component, but it can be derived from any `uvm_component`. The root node defines the test case to be executed by specifying the configuration of the testbench components and the stimulus to be executed by them.

For instance, in order to specify the test component 'my\_test' as the UVM testbench root class, the Questa command line would be:

```
vsim tb_top +UVM_TESTNAME=my_test
```

## Phase Descriptions

The following section describes the purpose of each of the different UVM phases

### Build Phases

The build phases are executed at the start of the UVM testbench simulation and their overall purpose is to construct, configure and connect the testbench component hierarchy.

All the build phase methods are functions and therefore execute in zero simulation time.

#### **build**

Once the UVM testbench root node component is constructed, the build phase starts to execute. It constructs the testbench component hierarchy from the top downwards. The construction of each component is deferred so that each layer in the component hierarchy can be configured by the level above. During the build phase `uvm_components` are indirectly constructed using the UVM factory.

#### **connect**

The connect phase is used to make TLM connections between components or to assign handles to testbench resources. It has to occur after the build method has put the testbench component hierarchy in place and works from the bottom of the hierarchy upwards.

#### **end\_of\_elaboration**

The `end_of_elaboration` phase is used to make any final adjustments to the structure, configuration or connectivity of the testbench before simulation starts. Its implementation can assume that the testbench component hierarchy and inter-connectivity is in place. This phase executes bottom up.

## Run Time Phases

The testbench stimulus is generated and executed during the run time phases which follow the build phases. After the `start_of_simulation` phase, the UVM executes the `run` phase and the phases `pre_reset` through to `post_shutdown` in parallel. The `run` phase was present in the OVM and is preserved to allow OVM components to be easily migrated to the UVM. It is also the phase that transactors will use. The other phases were added to the UVM to give finer run-time phase granularity for tests, scoreboards and other similar components. It is expected that most testbenches will only use `reset`, `configure`, `main` and `shutdown` and not their `pre` and `post` variants

#### **start\_of\_simulation**

The `start_of_simulation` phase is a function which occurs before the time consuming part of the testbench begins. It is intended to be used for displaying banners; testbench topology; or configuration information. It is called in bottom up order.

#### **run**

The `run` phase occurs after the `start_of_simulation` phase and is used for the stimulus generation and checking activities of the testbench. The `run` phase is implemented as a task, and all `uvm_component` `run` tasks are executed in parallel. Transactors such as drivers and monitors will nearly always use this phase.

**pre\_reset**

The pre\_reset phase starts at the same time as the run phase. Its purpose is to take care of any activity that should occur before reset, such as waiting for a power-good signal to go active. We do not anticipate much use for this phase.

**reset**

The reset phase is reserved for DUT or interface specific reset behaviour. For example, this phase would be used to generate a reset and to put an interface into its default state.

**post\_reset**

The post\_reset phase is intended for any activity required immediately following reset. This might include training or rate negotiation behaviour. We do not anticipate much use for this phase.

**pre\_configure**

The pre\_configure phase is intended for anything that is required to prepare for the DUT's configuration process after reset is completed, such as waiting for components (e.g. drivers) required for configuration to complete training and/or rate negotiation. It may also be used as a last chance to modify the information described by the test/environment to be uploaded to the DUT. We do not anticipate much use for this phase.

**configure**

The configure phase is used to program the DUT and any memories in the testbench so that it is ready for the start of the test case. It can also be used to set signals to a state ready for the test case start.

**post\_configure**

The post\_configure phase is used to wait for the effects of configuration to propagate through the DUT, or for it to reach a state where it is ready to start the main test stimulus. We do not anticipate much use for this phase.

**pre\_main**

The pre\_main phase is used to ensure that all required components are ready to start generating stimulus. We do not anticipate much use for this phase.

**main**

This is where the stimulus specified by the test case is generated and applied to the DUT. It completes when either all stimulus is exhausted or a timeout occurs. Most data throughput will be handled by sequences started in this phase.

**post\_main**

This phase is used to take care of any finalization of the main phase. We do not anticipate much use for this phase.

**pre\_shutdown**

This phase is a buffer for any DUT stimulus that needs to take place before the shutdown phase. We do not anticipate much use for this phase.

**shutdown**

The shutdown phase is used to ensure that the effects of the stimulus generated during the main phase have propagated through the DUT and that any resultant data has drained away. It might also be used to execute time consuming sequences that read status registers.

**post\_shutdown**

Perform any final activities before exiting the active simulation phases. At the end of the post\_shutdown phase, the UVM testbench execution process starts the clean up phases. We do not anticipate much use for this phase ( with the possible exception of some object-to-one code ).

## Clean Up Phases

The clean up phases are used to extract information from scoreboards and functional coverage monitors to determine whether the test case has passed and/or reached its coverage goals. The clean up phases are implemented as functions and therefore take zero time to execute. They work from the bottom to the top of the component hierarchy.

**extract**

The extract phase is used to retrieve and process information from scoreboards and functional coverage monitors. This may include the calculation of statistical information used by the report phase. This phase is usually used by analysis components.

**check**

The check phase is used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the testbench. This phase is usually used by analysis components.

**report**

The report phase is used to display the results of the simulation or to write the results to file. This phase is usually used by analysis components.

**final**

The final phase is used to complete any other outstanding actions that the testbench has not already completed.

# Factory

---

## The UVM Factory

The purpose of the UVM factory is to allow an object of one type to be substituted with an object of a derived type without having to change the structure of the testbench or edit the testbench code. The mechanism used is referred to as an override and the override can be by instance or type. This functionality is useful for changing sequence functionality or for changing one version of a component for another. Any components which are to be swapped must be polymorphically compatible. This includes having all the same TLM interface handles and TLM objects must be created by the new replacement component. Additionally, in order to take advantage of the factory certain coding conventions need to be followed.

### Factory Coding Convention 1: Registration

A component or object must contain factory registration code which comprises of the following elements:

- An `uvm_component_registry` wrapper, typedefed to `type_id`
- A static function to get the `type_id`
- A function to get the type name

For example:

```
class my_component extends uvm_component;

// Wrapper class around the component class that is used within the factory
typedef uvm_component_registry #(my_component, "my_component") type_id;

// Used to get the type_id wrapper
static function type_id get_type();
    return type_id::get();
endfunction

// Used to get the type_name as a string
function string get_type_name();
    return "my_component";
endfunction

...
endclass: my_component
```

The registration code has a regular pattern and can be safely generated with one of a set of four factory registration macros:

```
// For a component
class my_component extends uvm_component;

// Component factory registration macro
```

```

`uvm_component_utils(my_component)

// For a parameterised component

class my_param_component #(int ADD_WIDTH=20, int DATA_WIDTH=23) extends uvm_component;

typedef my_param_component #(ADD_WIDTH, DATA_WIDTH) this_t;

// Parameterised component factory registration macro

`uvm_component_param_utils(this_t)

// For a class derived from an object (uvm_object, uvm_transaction, uvm_sequence_item, uvm_sequence etc)

class my_item extends uvm_sequence_item;

`uvm_object_utils(my_item)

// For a parameterised object class

class my_item #(int ADD_WIDTH=20, int DATA_WIDTH=20) extends uvm_sequence_item;

typedef my_item #(ADD_WIDTH, DATA_WIDTH) this_t

`uvm_object_param_utils(this_t)

```

## Factory Coding Convention 2: Constructor Defaults

The uvm\_component and uvm\_object constructors are virtual methods which means that users have to follow their prototype template. In order to support deferred construction during the build phase, the factory constructor should contain defaults for the constructor arguments. This allows a factory registered class to be built inside the factory using the defaults and then the class properties are re-assigned to the arguments passed via the create method of the uvm\_component\_registry wrapper class. The defaults are different for components and objects:

```

// For a component

class my_component extends uvm_component;

function new(string name = "my_component", uvm_component parent = null);
  super.new(name, parent);
endfunction

// For an object

class my_item extends uvm_sequence_item;

function new(string name = "my_item");
  super.new(name);
endfunction

```

## Factory Coding Convention 3: Component and Object creation

Testbench components are created during the build phase using the create method of the uvm\_component\_registry. This first constructs the class, then assigns the pointer to the class to its declaration handle in the testbench after having the name and parent arguments assigned correctly. For components, the build process is top-down, which allows higher level components and configurations to control what actually gets built.

Object classes are created as required, again the create method is used.

The following code snippet illustrates how this is done:

```
class env extends uvm_env;

my_component m_my_component;
my_param_component #( .ADDR_WIDTH(32), .DATA_WIDTH(32) ) m_my_p_component;

// Constructor & registration macro left out

// Component and Parameterized Component create examples
function void build_phase( uvm_phase phase );
    m_my_component = my_component::type_id::create("m_my_component", this);
    m_my_p_component = my_param_component #(32, 32)::type_id::create("m_my_p_component", this);
endfunction: build

task run_phase( uvm_phase phase );
    my_seq test_seq;
    my_param_seq #( .ADDR_WIDTH(32), .DATA_WIDTH(32) ) p_test_seq;

    // Object and parameterised object create examples
    test_seq = my_seq::type_id::create("test_seq");
    p_test_seq = my_param_seq #(32, 32)::type_id::create("p_test_seq");
    // ....
endtask: run
```

# UsingFactoryOverrides

The UVM factory allows a class to be substituted with another class of a derived type when it is constructed. This can be useful for changing the behaviour of a testbench by substituting one class for another without having to edit or re-compile the testbench code. In order for factory override process to work there are a number of coding convention pre-requisites that need to be followed, these are explained in the article on the *UVM factory*.

The UVM factory can be thought of as a lookup table. W

The UVM factory allows a class to be substituted with another class of a derived type when it is constructed. This can be useful for changing the behaviour of a testbench by substituting one class for another without having to edit or re-compile the testbench code. In order for the factory override process to work, there are a number of coding convention pre-requisites that need to be followed, these are explained in the article on the *UVM factory*.

The UVM factory can be thought of as a lookup table. When "normal" component construction takes place using the `<type>::type_id::create("<name>", <parent>")` approach, what happens is that the type\_id is used to pick the factory component wrapper for the class, construct its contents and pass the resultant handle back again. The factory override changes the way in which the lookup happens so that looking up the original type\_id results in a different type\_id being used. Consequently, a handle to a different type of constructed object is returned. This technique relies on polymorphism which is the ability to be able to refer to derived types using a base type handle. In practice, an override will only work when a parent class is overridden by one of its descendants in the class extension hierarchy.

## Component Overrides

There are two types of component overrides in the UVM - type overrides and instance overrides.

### Component Type Overrides

A type override means that every time a component class type is created in a testbench hierarchy, a substitute type is created in its place. This applies to all instances of that component type. The method calls for this type of override are illustrated in the following code fragment:

```
//  
// Component type override example  
// -----  
  
// Colour parent class  
class colour extends uvm_component;  
  
`uvm_component_utils(colour)  
  
// etc  
endclass: colour  
  
// Red child class  
class red extends colour;  
  
`uvm_component_utils(red)
```

```

//etc
endclass: red

//
// Factory type override syntax is:
//
// <original_type>::type_id::set_type_override(<substitute_type>::get_type(), replace);
//
// Where replace is a bit which when ==1 enables the override of an existing override, otherwise
// the existing override is honoured.

// To override all instances of colour with red:
colour::type_id::set_type_override(red::get_type(), 1);

// This means that the following creation line returns a red, rather than a colour
pixel = colour::type_id::create("pixel", this);

```

Parameterized component classes can also be overridden, but care must be taken to ensure that the overriding class has the same parameter values as the class that is being overridden, otherwise they are not considered to be of related types:

```

//
// Type overrides for parameterised classes:
// ----

// Base class type
class bus_driver #(int BUS_WIDTH = 32) extends uvm_component;

`uvm_component_param_utils(bus_driver #(BUS_WIDTH))
// etc

endclass: bus_driver

// Derived class type
class bus_conductor #(int BUS_WIDTH = 32) extends bus_driver #(BUS_WIDTH);

`uvm_component_param_utils(bus_conductor #(BUS_WIDTH))
// etc

endclass: bus_conductor

// The parameterised type override needs to keep the parameterisation consistent

bus_driver #(64)::type_id::set_type_override(bus_conductor #(64)::get_type(), 1); // This will succeed

```

```
// Creation of a #(64) bus_driver results in a #(64) bus_conductor handle being returned:  
  
bus_person = bus_driver#(64)::type_id::create("bus_person", this);  
  
// Whereas creating a #(16) bus_driver results in a #(16) bus_driver handle being returned because  
// the matching type override is not found:  
  
bus_person = bus_driver#(16)::type_id::create("bus_person", this);  
  
// Similarly if a type override has non-matching parameters, then it will fail and return the original type  
  
bus_driver #(64)::type_id::set_type_override(bus_conductor #(32)::get_type(), 1); // Returns bus_driver #(64)
```

## Component Instance Overrides

A specific component instance can be overriden by specifying its position in the uvm component hierarchy. Again, this approach can be used with parameterised classes provided care is taken to match the parameterisation of the two class types involved in the override:

```
//  
// Component Instance Factory Override example  
// -----  
  
// Using red --> colour example from type override example  
//  
// Syntax for the instance override:  
//  
// <original_type>::type_id::set_inst_override(<substitute_type>::get_type(), <path_string>);  
//  
  
colour::type_id::set_inst_override(red::get_type(), "top.env.raster.spot");  
  
// And again for a parameterised type, the parameter values must match  
  
bus_driver #(64)::type_id::set_inst_override(bus_conductor #(64)::get_type(), "top.env.bus_agent.m_driver");
```

## Object Overrides

Objects or sequence related objects are generally only used with type overrides since the instance override approach relates to a position in the UVM testbench component hierarchy which objects do not take part in. However there is a coding trick which can be used to override specific "instances" of an object and this is explained in the article on *overriding sequences*.

The code for an object override follows the same form as the component override. When "normal" component construction takes place using the `<type>::type_id::create("<name>", <parent>")` approach, what happens is that the type\_id is used to pick the factory component wrapper for the class, construct its contents and pass the resultant handle back again. The factory override changes the way in which the lookup happens so that looking up the original type\_id results in a different type\_id being used and consequently a handle to a different type of constructed object being returned. This technique relies on polymorphism, in other words the ability to be able to refer to derived types using a base type handle. In practice, an override will only work when a parent class is overridden by one of its descendants in the class extension hierarchy.

## Component Overrides

There are two types of component overrides in the UVM - type overrides and instance overrides.

### Component Type Overrides

A type override means that every time a component class type is created in a testbench hierarchy, a substitute type is created in its place. This applies to all instances of that component type. The method calls for this type of override are illustrated in the following code fragment:

```
//
// Component type override example
// -----
// Colour parent class
class colour extends uvm_component;

`uvm_component_utils(colour)

// etc
endclass: colour

// Red child class
class red extends colour;

`uvm_component_utils(red)

//etc
endclass: red

//
// Factory type override syntax is:
```

```

// 
// <original_type>::type_id::set_type_override(<substitute_type>::get_type(), replace);
// 
// Where replace is a bit which when ==1 enables the override of an existing override, otherwise
// the existing override is honoured.

// To override all instances of colour with red:
colour::type_id::set_type_override(red::get_type(), 1);

// This means that the following creation line returns a red, rather than a colour
pixel = colour::type_id::create("pixel", this);

```

Parameterised component classes can also be overridden, but care must be taken to ensure that the overriding class has the same parameter values as the class that is being overridden, otherwise they are not considered to be of related types:

```

// 
// Type overrides for parameterised classes:
// ----

// Base class type
class bus_driver #(int BUS_WIDTH = 32) extends uvm_component;

`uvm_component_param_utils(bus_driver #(BUS_WIDTH))
// etc

endclass: bus_driver

// Derived class type
class bus_conductor #(int BUS_WIDTH = 32) extends bus_driver #(BUS_WIDTH);

`uvm_component_param_utils(bus_conductor #(BUS_WIDTH))
// etc

endclass: bus_conductor

// The parameterised type override needs to keep the parameterisation consistent

bus_driver #(64)::type_id::set_type_override(bus_conductor #(64)::get_type(), 1); // This will succeed

// Creation of a #(64) bus_driver results in a #(64) bus_conductor handle being returned:

bus_person = bus_driver#(64)::type_id::create("bus_person", this);

// Whereas creating a #(16) bus_driver results in a #(16) bus_driver handle being returned because
// the matching type override is not found:

```

```
bus_person = bus_driver #(16)::type_id::create("bus_person", this);

// Similarly if a type override has non-matching parameters, then it will fail and return the original type

bus_driver #(64)::type_id::set_type_override(bus_conductor #(32)::get_type(), 1); // Returns bus_driver #(64)
```

## Component Instance Overrides

A specific component instance can be overriden by specifying its position in the uvm component hierarchy. Again, this approach can be used with parameterised classes provided care is taken to match the parameterisation of the two class types involved in the override:

```
//  
// Component Instance Factory Override example  
// -----  
  
// Using red --> colour example from type override example  
//  
// Syntax for the instance override:  
//  
// <original_type>::type_id::set_inst_override(<substitute_type>::get_type(), <path_string>);  
//  
  
colour::type_id::set_inst_override(red::get_type(), "top.env.raster.spot");  
  
// And again for a parameterised type, the parameter values must match  
  
bus_driver #(64)::type_id::set_inst_override(bus_conductor #(64)::get_type(), "top.env.bus_agent.m_driver");
```

## Object Overrides

Objects or sequence related objects are generally only used with type overrides since the instance override approach relates to a position in the UVM testbench component hierarchy which objects do not take part in. However there is a coding trick which can be used to override specific "instances" of an object and this is explained in the article on *overriding sequences*.

The code for an object override follows the same form as the component override.

# SystemVerilog Packages

---

A package is a SystemVerilog language construct that enables related declarations and definitions to be grouped together in a package namespace. A package might contain type definitions, constant declarations, functions and class templates. To use a package within a scope, it must be imported, after which its contents can be referenced.

The package is a useful means to organize code and to make sure that references to types, classes etc are consistent. The UVM base class library is contained within one package called the "uvm\_pkg". When developing UVM testbenches packages should be used to collect together and organize the various class definitions that are developed to implement agents, envs, sequence libraries, test libraries etc.

## UVM Package Coding Guidelines

### **Package naming and file naming conventions:**

A package should be named with a \_pkg post-fix. The name of the file containing the package should reflect the name of the package and have a .sv extension.

**For example:** The file spi\_env\_pkg.sv would contain the package spi\_env\_pkg.

**Justification:** The .sv extension is a convention that denotes that the package file is a stand-alone compilation unit. The \_pkg post fix denotes that the file contains a package. Both of these conventions are useful to humans and machine parsing scripts.

### **Classes contained within a package should be `included**

Class templates that are declared within the scope of a package should be separated out into individual files with a .svh extension. These files should be `included in the package in the order in which they need to be compiled. The package file is the only place where `includes should be used, there should be no further `include statements inside the included files.

**Justification:** Having the classes declared in separate files makes them easier to maintain, and it also makes it clearer what the package content is.

### **Imports from other packages should be declared at the head of the package**

A package's content may need to refer to the contents of another package. In this case the external packages should be declared at the start of the package code body. Individual files, such as class templates that may be included, should not do separate imports.

**Justification:** Grouping all the imports in one place makes it clear what the dependencies of the package are. Placing imports in other parts of the package or inside included files can cause ordering and potential type clashes.

### All the files used by a package should be collected together in one directory

All of the files to be included in a package should be collected together in a single directory. This is particularly important for agents where the agent directory structure needs to be a complete stand-alone package.

**Justification:** This makes compilation easier since there is only one include directory, it also helps with reuse since all the files for a package can be collected together easily.

Below is an example of a package file for an UVM env. This env contains two agents (spi and apb) and a register model and these are imported as sub-packages. The class templates relevant to the env are `included:

```
// Note that this code is contained in a file called spi_env_pkg.sv
//
// In Questa it would be compiled using:
// vlog +incdir+$UVM_HOME/src+<path_to_spi_env> <path_to_spi_env>/spi_env_pkg.sv
//

//
// Package Description:
//
package spi_env_pkg;

// Standard UVM import & include:
import uvm_pkg::*;
`include "uvm_macros.svh"

// Any further package imports:
import apb_agent_pkg::*;
import spi_agent_pkg::*;
import spi_register_pkg::*;

// Includes:
`include "spi_env_config.svh"
`include "spi_virtual_sequencer.svh"
`include "spi_env.svh"

endpackage: spi_env_pkg
```

## Package Scopes

Something that often confuses users is that the SystemVerilog package is a scope. This means that everything declared within a package, and the contents of other packages imported into the package are only visible within the scope of the package. If a package is imported into another scope (i.e. another package or a module) then only the contents of the package are visible and not the contents of any packages that it imported. If the content of these other packages are needed in the new scope, then they need to be imported separately.

```
//
// Package Scope Example
```

```
// -----
// 
package spi_test_pkg;

// The UVM package has to be imported, even though it is imported
// in the spi_env package. This is because the import of the uvm_pkg
// is only visible within the current scope
import uvm_pkg::*;

// The same is true of the `include of the uvm_macros
`include "uvm_macros.svh"

// Import of uvm_pkg inside the spi_env package is not
// visible within the scope of the spi_test package
import spi_env_pkg::*;

// Other imports
// Other `includes
`include spi_test_base.svh

endpackage: spi_test_pkg
```

# Connections to DUT Interfaces

## Connections

Learn about DUT Interface Connections, techniques for hookup and reuse

### Connections Chapter contents:

Connections (this page)

SVCreationOrder

Connect/SystemVerilogTechniques

ParameterizedTests

    Configuration

    Config/Params Package

Connect/Virtual Interface

    Connect/VirtInterfaceConfigPkg

        Connect/VirtInterfacePackage

        VirtInterfaceFunctionCallChain

    BusFunctionalModels

DualTop

ProtocolModules

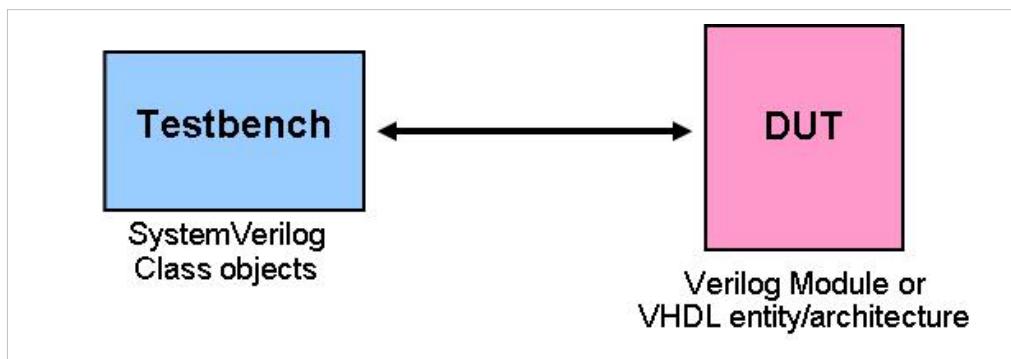
Connect/AbstractConcrete

    Connect/TwoKingdomsFactory

## Topic Overview

### Introduction

The Device Under Test (DUT) is typically a Verilog module or a VHDL entity/architecture while the testbench is composed of SystemVerilog class objects.

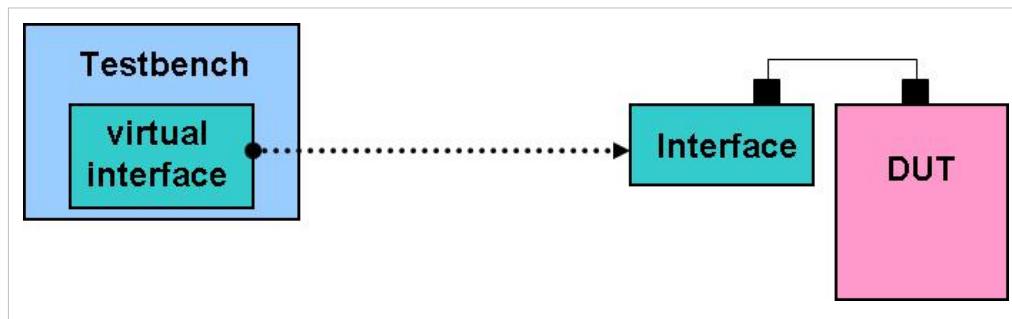


There are a number of factors to consider in DUT - Testbench (TB) connection and communication; module instance to class object communication mechanisms, configuration of the DUT, reuse, emulation, black box/white box testing and so forth. There are quite a number of different approaches and solutions for managing the different pieces of this puzzle. The challenge is to manage it in a way that addresses all these different factors.

## DUT-TB Communication

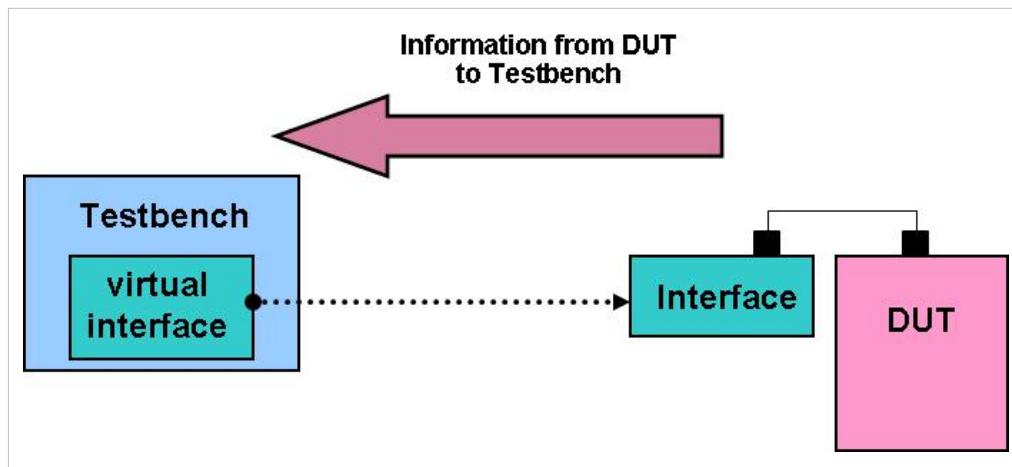
The DUT and testbench belong to two different SystemVerilog instance worlds. The DUT belongs to the static instance world while the testbench belongs to the dynamic instance world. Because of this the DUT's ports can not be connected directly to the testbench class objects so a different SystemVerilog means of communication, which is virtual interfaces, is used.

The DUT's ports are connected to an instance of an interface. The Testbench communicates with the DUT through the interface instance. Using a virtual interface as a reference or handle to the interface instance, the testbench can access the tasks, functions, ports, and internal variables of the SystemVerilog interface. As the interface instance is connected to the DUT pins, the testbench can monitor and control the DUT pins indirectly through the interface elements.

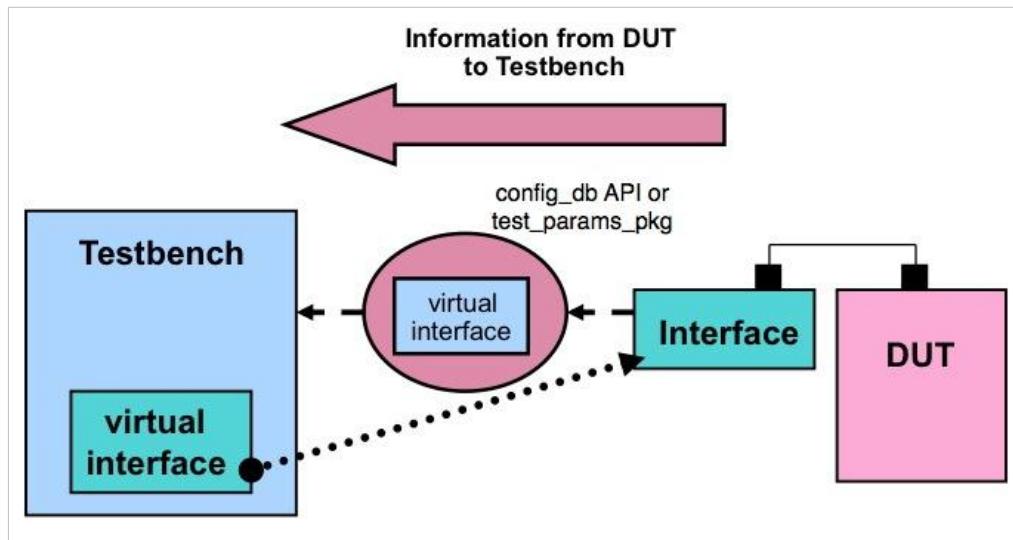


Sometimes a virtual interface approach cannot be used. In which case there is a second or alternative approach to DUT-TB communication which is referred to as the abstract/concrete class approach that may be used. However, as long as it can be used, virtual interfaces is the preferred and recommended approach.

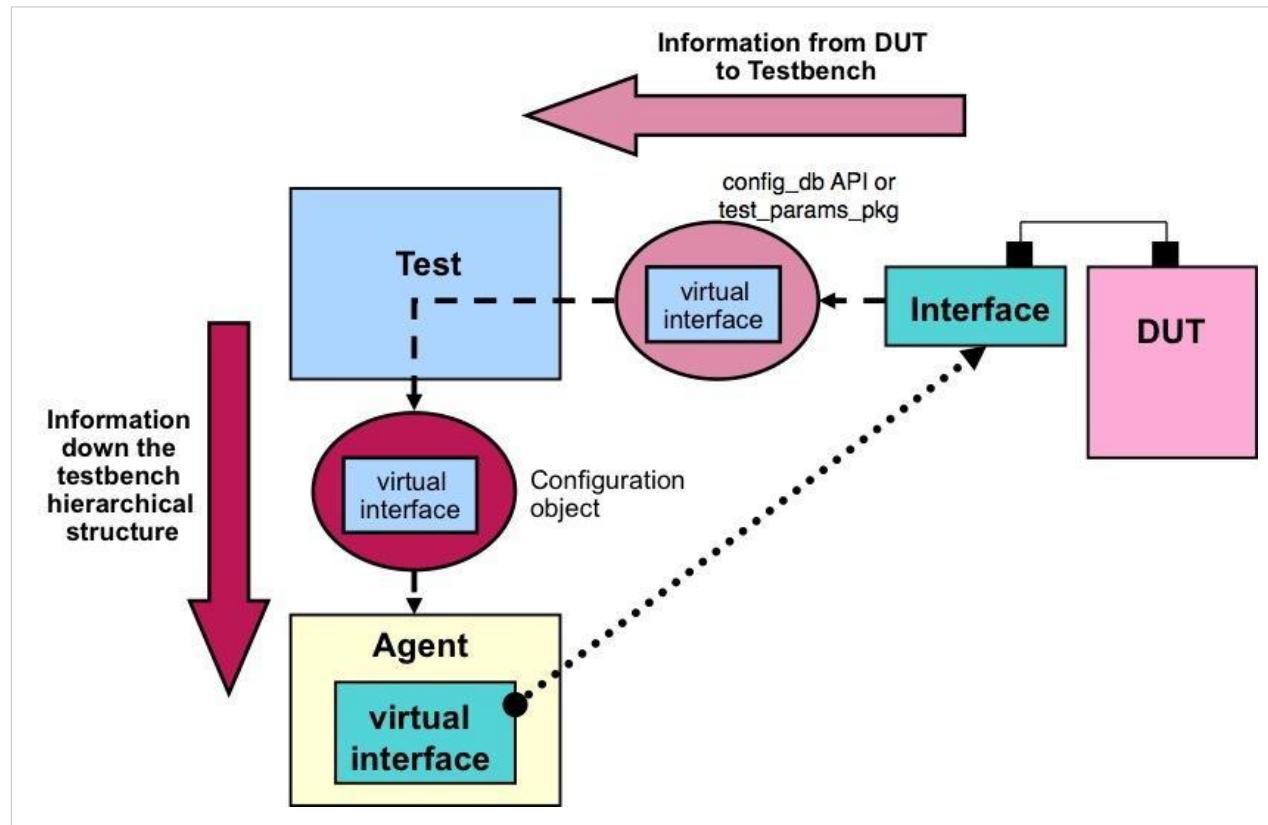
Regardless of which approach is used, instance information must be passed from the DUT to the testbench.



When using virtual interfaces, the location of the interface instance is supplied to the testbench so its virtual interface properties may be set to point to the interface instance. The recommended approach for passing this information to the testbench is to use either the configuration database using the config\_db API or to use a package.



The test class in the testbench receives the information on the location of the interface instance. After receiving this information, the test class supplies this information to the agent transactors that actually need the information. The test class does this by placing the information in a configuration object which is provided to the appropriate agent.



More detailed discussion and examples of passing virtual interface information to the testbench from the DUT and on setting virtual interfaces for DUT-TB communication is in the article on virtual interfaces.

## DUT-TB Configuration

### Parameter sharing between the DUT and Testbench

When the DUT and/or the associated interface(s) are parameterized, the parameter values are almost always used in the testbench as well. These common parameters should be defined in a single location and then shared with both the DUT and the testbench. The recommended way to do this is to place in a package the parameters that are used both in the DUT and testbench. This package is referred to as the test parameters package.

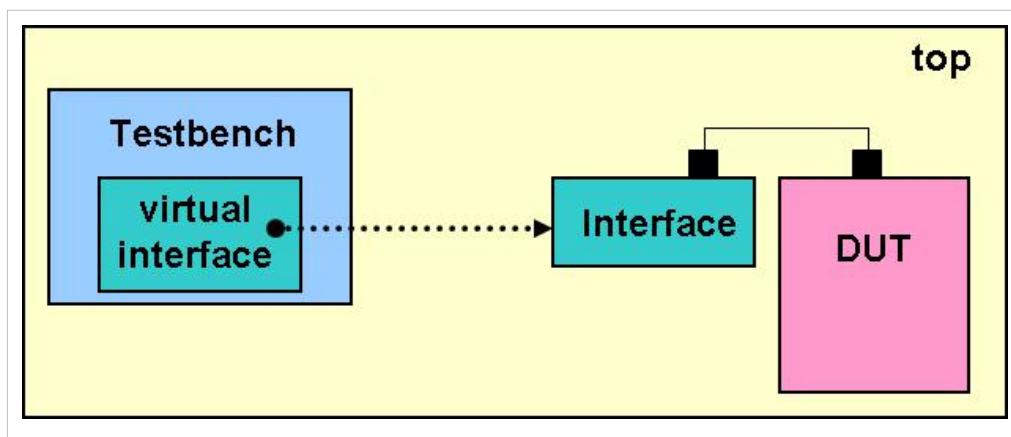
The test parameter package may also be used to pass the location of the interface instance from the DUT to the testbench as explained earlier. There is an example and more detailed explanation in the article on "setting virtual interface properties in the testbench with the configuration dataset using the test parameter package"

### Parameterized Tests

Another approach to passing parameters into the testbench is to parameterize the top level class in the testbench which is typically the test. There are a number of issues with parameterized tests that are discussed along with solutions.

### Encapsulation

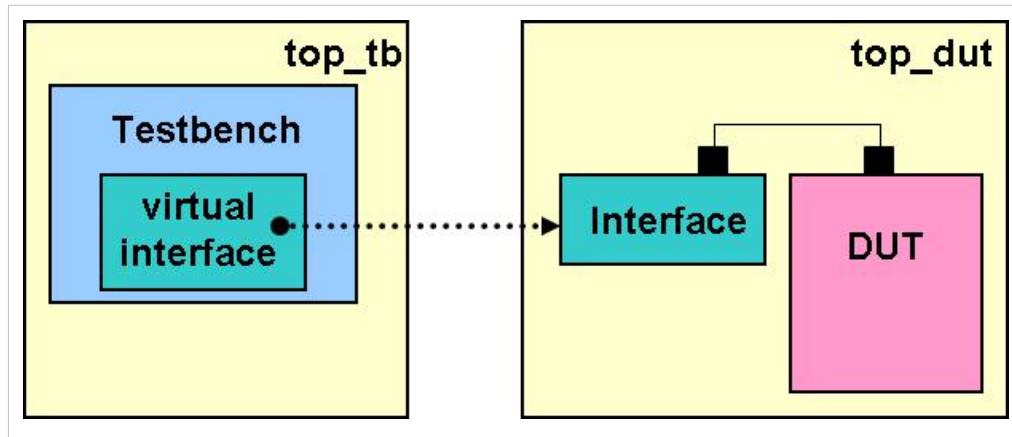
A typical DUT-TB setup has a top level SystemVerilog module that is a container for both the testbench and the DUT with its associated connection and support logic (such as clock generation). This style setup is referred to as a single top



The top level module can become messy, complicated and hard to manage. When this occurs it is recommended to group items by encapsulating them inside of wrapper modules. Encapsulation also provides for modularity for swapping and for reuse. Several different levels of encapsulation may be considered and are discussed below.

### Dual Top

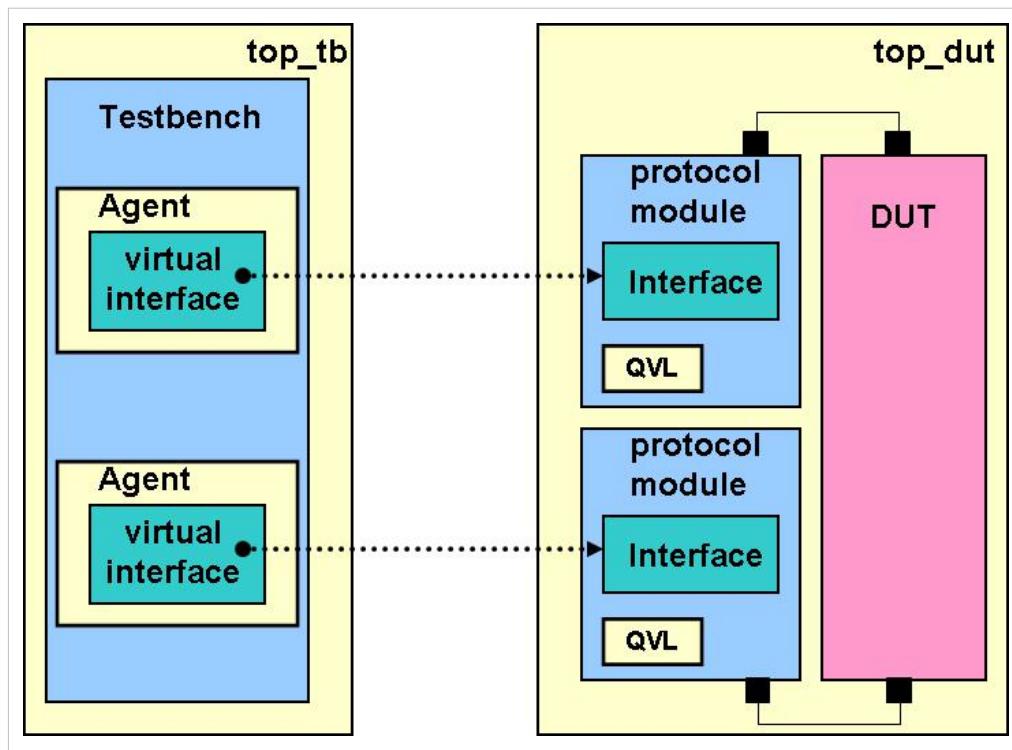
A level of encapsulation where two top modules are used is called dual top. One of the top modules is a DUT wrapper module that includes the DUT, interfaces, protocol modules, clock generation logic, DUT wires, registers etc. The other top module is a wrapper module which creates the testbench. When emulation is a consideration, Dual top is a necessity. The DUT wrapper is the code that goes in the emulator. The testbench wrapper module stays running in the simulator. If the testbench is only going to be used in simulation, dual top is not necessary but may however still provide a useful level of encapsulation for modularity, reuse etc.



The passing of information from the DUT to the testbench is the same as described earlier. A more detailed explanation and example is in the article Dual Top.

### Protocol Modules

When emulation is a consideration, another level of encapsulation called protocol modules is necessary to isolate the changes that occur in the agent and interface in moving between simulation and emulation. Protocol modules are wrapper modules that encapsulate a DUT interface, associated assertions, QVL instances (which are not allowed inside an interface), and so forth.



If the testbench is only going to be used in simulation, protocol modules are not necessary. They may however still provide a useful level of encapsulation for modularity, reuse etc.

## **Blackbox and Whitebox Testing**

### **Blackbox testing**

Blackbox testing of the DUT is a method of testing that tests the functionality of the DUT at the interface or pins of the DUT without specific knowledge of or access to the DUT's internal structure. The writer of the test selects valid and invalid inputs and determines the correct response. Black box access to the DUT is provided typically by a virtual interface connection to an interface instance connected to the pins of the DUT.

### **Whitebox testing**

Whitebox testing of the DUT is a method of testing that tests the internal workings of the DUT. Access to the DUT's internal structure is required. Providing this access effects the structure of the DUT-TB communication and must be taken into account if white box testing is a requirement.

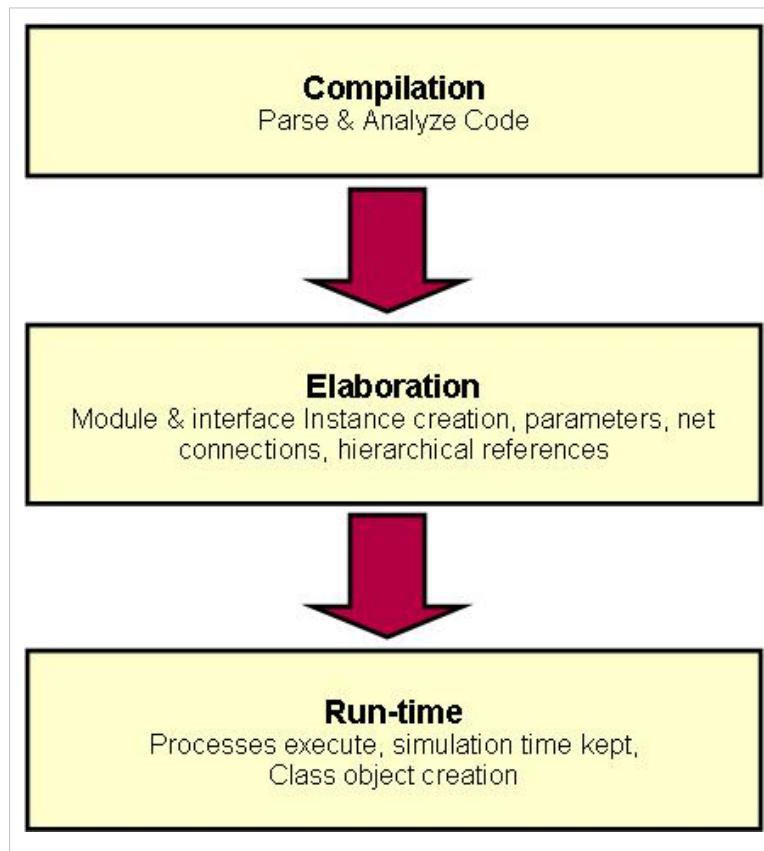
# SVCreationOrder

## SystemVerilog Instance Worlds

When generating an UVM testbench and in particular the DUT - testbench communication, it is helpful to understand the differences between the two different "instance worlds" of SystemVerilog and the order in which things are created.

## SystemVerilog Simulation Steps or Phases

A SystemVerilog simulation consists of three steps or phases (not to be confused with UVM phases): compilation, elaboration and run-time.



**Compilation** is where the code is parsed and analyzed.

**Elaboration** is the process of binding together the components of the design. Elaboration includes, among other things, creating instantiations, computing parameter values, resolving hierarchical names and connecting nets. Often when referring to the compilation and elaboration phases they are not distinguished but are generally referred to as compilation. In other words a "compile time error" may refer to an error at any time prior to run-time.

**Run-time** is what is thought of as the simulation actually executing or running with processes executing, simulation time advances etc. This step or phase is often referred to as "simulation".

Phrases such as "prior to simulation" or "before simulation" are often used to refer to the compilation and elaboration steps that happen before run-time or "during simulation" to refer to the run-time step or phase.

## Static Instance World

Many SystemVerilog component instances are created during elaboration before the simulation begins. Once simulation begins instances of these components are neither created nor destroyed but remain throughout the simulation. We refer to this as the static instance world. Components that belong to this world are module instances, interface instances, checker instances, primitive instances and the top level of the design hierarchy.

## Dynamic Instance World

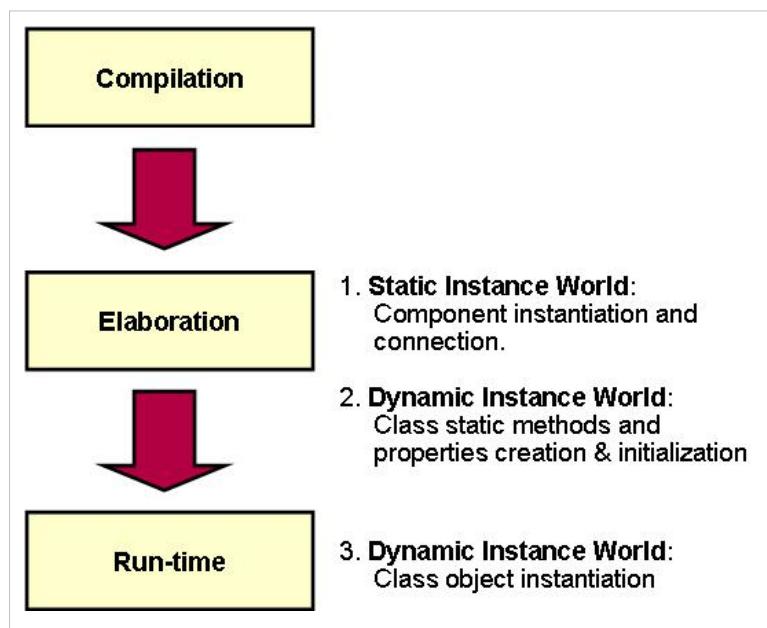
Component instances that may be created and destroyed during simulation (the SystemVerilog run phase) belong to what is referred to as the dynamic instance world. Components that belong to this world are classes.

There is an exception to this however. Class methods and properties that are declared as static are created prior to runtime. They are, however, created after the component instances of the static world.

This exception is often used to create and initialize class properties (including class objects) before simulation. This is referred to as static initialization. The UVM factory is an example of an object that is statically initialized.

## Order of Creation

The components of the two instance worlds are created in this order:



During Elaboration:

1. Component instances of the static world
2. Static methods and static properties of classes

During run-time:

1. Class instances

# Connect/SystemVerilogTechniques

---

## Introduction and Recommendations

SystemVerilog provides in general four different means of communication or connection between instances: ports, pointers, Verilog hierarchical paths, and shared variables. For class based testbenches ports may not be used. Hierarchical paths are not recommended. Pointers are the common means used. Shared variables may be used in limited areas.

## Ports

Ports are connections between members of the Static Instance World such as module and interface instances. Therefore they *may not* be used in classes which are part of the Dynamic Instance World.

UVM provides a notion of ports such as `uvm_tlm_put_port` etc. These are not SystemVerilog ports but rather are wrapper classes around pointers. Hence a UVM TLM port is a pointer based communication scheme dressed up like ports to look familiar to Verilog and VHDL engineers.

## Handles

A class handle is what points to a class object (instance). It is called a handle to differentiate from a pointer. A handle is what is considered a safe-pointer because of the restrictive rules of use compared to pointers in other languages such as C.

A virtual interface is a variable that represents an interface instance. It may be thought of as a handle to an interface instance.

## Verilog Hierarchical Path

Every identifier in SystemVerilog has a unique hierarchical path name. Any named SystemVerilog object can be referenced by this hierarchical path name from any other SystemVerilog object.

While this is powerful as it allows for communication from anywhere to anywhere, it is in general not desirable to use this technique. In modeling it is an absolute "no-no" because a connection is implicit in a hierarchical reference. This implicit connection won't be synthesized or created when the model is implemented, thus breaking the design. In verification it is to be avoided because portability and reuse are limited when it is used.

## Shared Variables

Shared variables are sometimes referred to as global variables although generally speaking they are not truly global in scope. A shared variable is a variable declared in a scope that may be referenced by other scopes. In shared variable behavior, the variable may be read and or written in these other scopes. The two most common examples of shared variables used in testbenches are variables declared in packages and static property declarations of classes.

In packages a variable may be declared such as an int or virtual interface. These variables may be referenced (i.e. both read and written) within other scopes such as classes or modules either by a fully resolved name (`package_name::variable_name`) or by an import.

Static property declarations of classes may be referenced by a fully resolved name (`class_name::static_property_name`). Often a static method of a class may be provided for accessing the static property.

It is recommended that shared variables only be used for initialization or status type communication where there is a

clear relationship between when information is written and read.

Shared variables are not recommended for the transfer of data, such as transactions, between objects.

Because of the nature of shared variables race conditions are inherent and so care must be taken or races will occur.

# ParameterizedTests

---

## Introduction

When configuring a test environment, there are two situations where SystemVerilog parameters are the only option available - type parameters and parameters used to specify bit vector sizes. Due to the nature of SystemVerilog parameters, the latest time that these values can be set is at elaboration time, which is usually at the point where you invoke the simulator (See regression test performance below).

## DVCon 2011 Paper

The information in this article was also presented at DVCon 2011 with Xilinx. The DVCon paper is available for download (Parameters And OVM - Can't They Just Get Along? [1]). The material in this article is a result of a collaboration between Mentor and Xilinx.

## Registering Parameterized classes with the String-based Factory

Parameterized classes use the `uvm\_component\_param\_utils and `uvm\_object\_param\_utils macros to register with the factory. There are actually two factories, however - one string-based and one type-based. The param\_utils macros only register with the type-based factory.

Occasionally, you might want to use the string-based factory to create a component or object. The most common case where the string-based factory is used is during the call to run\_test(). run\_test() uses either its string argument or the string value from the UVM\_TESTNAME plusarg to request a component from the string-based factory.

Since a parameterized component does not register with the string-based factory by default, you will need to create a string-based registration for your top-level test classes so that they can be instantiated by run\_test().

To accomplish this, you need to manually implement the actions that the param\_utils macro performs.

For example, given a parameterized test class named alu\_basic\_test #(DATA\_WIDTH), the macro call `uvm\_component\_param\_utils(alu\_basic\_test #(DATA\_WIDTH)) would expand to:

```
typedef uvm_component_registry #(alu_basic_test #(DATA_WIDTH)) type_id;

static function type_id get_type();
    return type_id::get();
endfunction

virtual function uvm_object_wrapper get_object_type();
    return type_id::get();
endfunction
```

The typedef in the code above creates a specialization of the uvm\_component\_registry type, but that type takes two parameter arguments - the first is the type being registered (alu\_basic\_test #(DATA\_WIDTH) in this case) with the type-based factory, and the second is the string name that will be used to uniquely identify that type in the string-based registry. Since the param\_utils macro does not provide a value for the second parameter, it defaults to the null string and no string-based registration is performed.

To create a string-based registration, you need to provide a string for the second parameter argument that will be unique for each specialization of the test class. You can rewrite the typedef to look like:

```
typedef uvm_component_registry #(alu_basic_test #(DATA_WIDTH), "basic_test1") type_id;
```

In addition, you would need to declare a "dummy" specialization of the parameterized test class so that the string name specified above is tied to the particular parameter values.

```
module testbench #(int DATA_WIDTH);

    alu_rtl #(DATA_WIDTH) dut ( /* port connections */ );

    // Associate the string "basic_test1" with the value of DATA_WIDTH
    typedef alu_basic_test #(DATA_WIDTH) dummy;

    initial begin
        run_test("basic_test1");
    end

endmodule
```

Note: instead of a name like "basic\_test1", you could use the macro described below to generate a string name like "basic\_test\_#(8)" with the actual parameter values as part of the string.

## Regression Test Performance with Parameters

In order to increase simulation performance, QuestaSim performs some elaboration tasks, including specifying top-level parameters, in a separate optimization step via the **vopt** tool. This tool takes top-level parameters and "bakes" them into the design in order to take full advantage of the design structure for optimization.

Unfortunately, this means that if you want to change the values of these parameters, it requires a re-optimization of the entire design before invocation of the simulator. This could have a significant impact on regression test performance where many test runs are made with different parameter values. To avoid this, you can tell **vopt** that certain parameters should be considered "floating", and will be specified later, by using the command-line options **+floatparameters** (for Verilog) and **+floatgenerics** (for VHDL).

Once the parameters have been specified as floating, you can use the **-g** option in **vsim** to set the parameter value. Subsequent changes to the parameter value only require re-invocation of **vsim** with a new value and do not require a re-optimization of the design.

The trade-off, however, is that these parameter values will not be used to help optimize the design for the best run-time performance. So, it is recommended that you use this technique sparingly, only when necessary (e.g. when the time cost of optimization is a measurable percentage of the total simulation run time). If necessary, you can separately pre-optimize the design with several parameter values, then select the optimization to use at run time.

## Maintaining Consistency in Parameter Lists

Many SystemVerilog parameters can naturally be grouped together in a conceptual "parameter list". These parameters tend to be declared together and are used in many places in the test environment.

Any change to a parameter list (e.g. adding a new parameter) usually requires careful editing of many different classes in many different files, and is an error-prone process.

An optional technique that can be employed is to create a set of macros that can reduce the chance of errors and enforce consistency.

<b>Declaration</b>	<code>`define params_declare #(int BUS_WIDTH = 16, int ADDR_WIDTH = 8)</code>
<b>Instantiation / Mapping</b>	<code>`define params_map #(.BUS_WIDTH(BUS_WIDTH), .ADDR_WIDTH(ADDR_WIDTH) )</code>
<b>String Value</b>	<code>`define params_string \$sformat("#(%ld, %ld)", BUS_WIDTH, ADDR_WIDTH)</code>

These macros keep with the reuse philosophy of minimizing areas of change. By using the macros, there is one, well-defined place to make changes in parameter lists.

# Connect/Virtual Interface

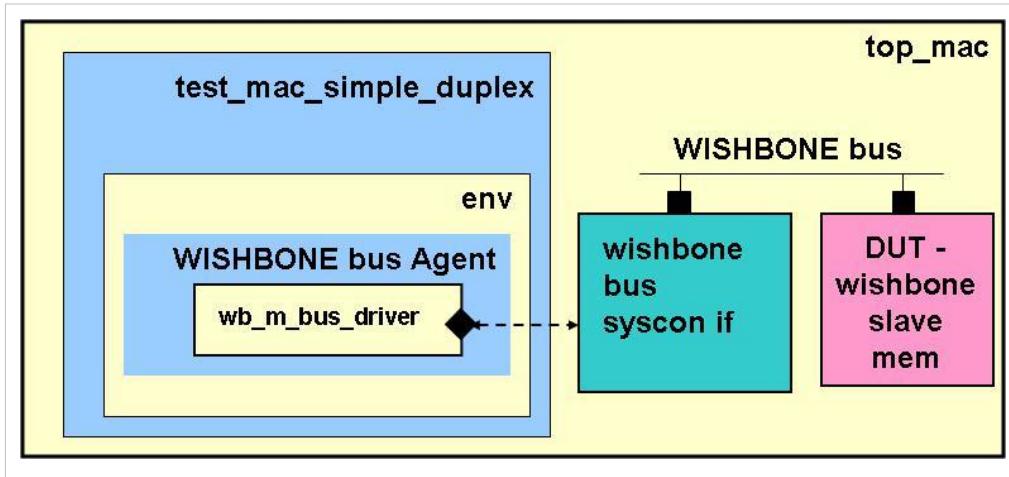
## Virtual Interfaces

A virtual interface is a dynamic variable that contains a reference to a static interface instance. For all intents and purposes, it can be thought of as a handle or reference to a SystemVerilog interface instance. Note that the use of the term "virtual" here is not used in the same sense as is it conventionally used in object oriented programming, but rather it is what the IEEE 1800 committee chose to call these references.

## Virtual Interface Use

Virtual interfaces are used because a class based testbench can not reference verilog/vhdl ports and hence cannot directly connect to a Verilog or VHDL DUT. Instead a SystemVerilog interface instance is connected to the ports of the DUT and the testbench communicates with the DUT through the interface instance. Using a virtual interface as a reference or handle to the interface instance, the testbench can access the tasks, functions, ports, and internal variables of the SystemVerilog interface. As the interface instance is connected to the DUT pins, the testbench can monitor and control the DUT pins indirectly through the interface elements.

The DUT may have one or more interfaces connected to its ports. If a DUT has multiple logical interfaces (different buses etc.), it is recommended to have a separate interface for each logical interface. Consideration should also be given to encapsulate the interface instances inside a protocol module.



An example DUT (WISHBONE bus slave memory in diagram) has the following ports:

```
module wb_slave_mem #(parameter mem_size = 13)
  (clk, rst, adr, din, dout, cyc, stb, sel, we, ack, err, rty);

  input      clk, rst;
  input [31:0] adr, din;
  output [31:0] dout;
  input      cyc, stb;
  input [3:0]  sel;
  input      we;
  output     ack, err, rty;
```

```

  ...
endmodule

```

In the WISHBONE bus environment there are a number of parameters that are shared between the DUT and the testbench. They are defined in a *test parameters package* (test\_params\_pkg) shown below. Of interest here are the mem\_slave\_size and mem\_slave\_wb\_id parameters. The mem\_slave\_size is used to set the size of the slave memory device. The WISHBONE bus has both masters and slaves with each having master and slave ids respectively. The mem\_slave\_wb\_id is used to set the WISHBONE slave id of the slave memory.

```

package test_params_pkg;
import uvm_pkg::*;

// WISHBONE general slave parameters
parameter slave_addr_space_sz = 32'h00100000;

// WISHBONE slave memory parameters
parameter mem_slave_size = 18; // 2**slave_mem_size = size in words(32 bits) of wb slave memory
parameter mem_slave_wb_id = 0; // WISHBONE bus slave id of wb slave memory
...

endpackage

```

A WISHBONE bus interconnect interface to connect to this DUT is below. This interconnect supports up to 8 masters and 8 slaves. Not shown here is the arbitration, clock, reset and slave decode logic. Only shown is the interconnect variables. A link to the full source is further down in this article.

```

// Wishbone bus system interconnect (syscon)
// for multiple master, multiple slave bus
// max 8 masters and 8 slaves

interface wishbone_bus_syscon_if #(int num_masters = 8, int num_slaves = 8,
                                         int data_width = 32, int addr_width = 32) ();

  // WISHBONE common signals
  bit clk;
  bit rst;
  bit [7:0] irq;

  // WISHBONE master outputs
  logic [data_width-1:0] m_wdata[num_masters];
  logic [addr_width-1:0] m_addr [num_masters];
  bit m_cyc [num_masters];
  bit m_lock[num_masters];
  bit m_stb [num_masters];
  bit m_we  [num_masters];
  bit m_ack [num_masters];
  bit [7:0] m_sel[num_masters];

```

```

// WISHBONE master inputs
bit m_err;
bit m_rty;
logic [data_width-1:0] m_rdata;

// WISHBONE slave inputs
logic [data_width-1:0] s_wdata;
logic [addr_width-1:0] s_addr;
bit [7:0] s_sel;
bit s_cyc;
bit s_stb[num_slaves]; //only input not shared since it is the select
bit s_we;

// WISHBONE slave outputs
logic [data_width-1:0] s_rdata[num_slaves];
bit s_err[num_slaves];
bit s_rty[num_slaves];
bit s_ack[num_slaves];
...
endinterface

```

To connect the interface to the DUT a hierarchical connection from the pins of the DUT to the variables in the interface is made as shown below. Note that the mem\_slave\_wb\_id parameter from the test\_params\_pkg is used as a slave "slot id" to connect the slave memory to the correct signals in the interface.

```

module top_mac;
import uvm_pkg::*;
import tests_pkg::*;
import uvm_container_pkg::*;
import test_params_pkg::*;

// WISHBONE interface instance
// Supports up to 8 masters and up to 8 slaves
wishbone_bus_syscon_if wb_bus_if();

//-----
// WISHBONE 0, slave 0: 000000 - 0fffff
// this is 1 Mbytes of memory
wb_slave_mem #(mem_slave_size) wb_s_0 (
  // inputs
  .clk ( wb_bus_if.clk ),
  .rst ( wb_bus_if.rst ),
  .adr ( wb_bus_if.s_addr ),
  .din ( wb_bus_if.s_wdata ),

```

```

    .cyc ( wb_bus_if.s_cyc ),
    .stb ( wb_bus_if.s_stb[mem_slave_wb_id] ),
    .sel ( wb_bus_if.s_sel[3:0] ),
    .we ( wb_bus_if.s_we ),
    // outputs
    .dout( wb_bus_if.s_rdata[mem_slave_wb_id] ),
    .ack ( wb_bus_if.s_ack[mem_slave_wb_id] ),
    .err ( wb_bus_if.s_err[mem_slave_wb_id] ),
    .rty ( wb_bus_if.s_rty[mem_slave_wb_id] )
  );
...
endmodule

```

In the testbench, access to the DUT is typically required in transactors such as drivers and monitors that reside in an agent. Assume in the code example below that the virtual interface property `m_v_wb_bus_if` points to the instance of the `wishbone_bus_syscon_if` connected to the DUT (the next section discusses setting the virtual interface property). Then in a WISHBONE bus driver the code might look like this. Note the use of the virtual interface property to access the interface variables :

```

class wb_m_bus_driver extends uvm_driver #(wb_txn, wb_txn);
  ...
  uvm_analysis_port #(wb_txn) wb_drv_ap;
  virtual wishbone_bus_syscon_if m_v_wb_bus_if; // Virtual Interface
  bit [2:0] m_id; // Wishbone bus master ID
  wb_config m_config;
  ...

  //WRITE 1 or more write cycles
  virtual task wb_write_cycle(ref wb_txn req_txn);
    wb_txn orig_req_txn;
    $cast(orig_req_txn, req_txn.clone()); //save off copy of original req transaction
    for(int i = 0; i<req_txn.count; i++) begin
      if(m_v_wb_bus_if.rst) begin
        reset(); // clear everything
        return; //exit if reset is asserted
      end
      m_v_wb_bus_if.m_wdata[m_id] = req_txn.data[i];
      m_v_wb_bus_if.m_addr[m_id] = req_txn.adr;
      m_v_wb_bus_if.m_we[m_id] = 1; //write
      m_v_wb_bus_if.m_sel[m_id] = req_txn.byte_sel;
      m_v_wb_bus_if.m_cyc[m_id] = 1;
      m_v_wb_bus_if.m_stb[m_id] = 1;
      @ (posedge m_v_wb_bus_if.clk)
      while (! (m_v_wb_bus_if.m_ack[m_id] & m_v_wb_bus_if.gnt[m_id])) @ (posedge m_v_wb_bus_if.clk);
      req_txn.adr = req_txn.adr + 4; // byte address so increment by 4 for word addr
    end
  endtask
endclass

```

```
end

`uvm_info($sformatf("WB_M_DRV_R_%0d",m_id),
           $sformatf("req_txn: %s",orig_req_txn.convert2string()),
           351 )

wb_drv_ap.write(orig_req_txn); //broadcast original transaction
m_v_wb_bus_if.m_cyc[m_id] = 0;
m_v_wb_bus_if.m_stb[m_id] = 0;

endtask

...
endclass
```

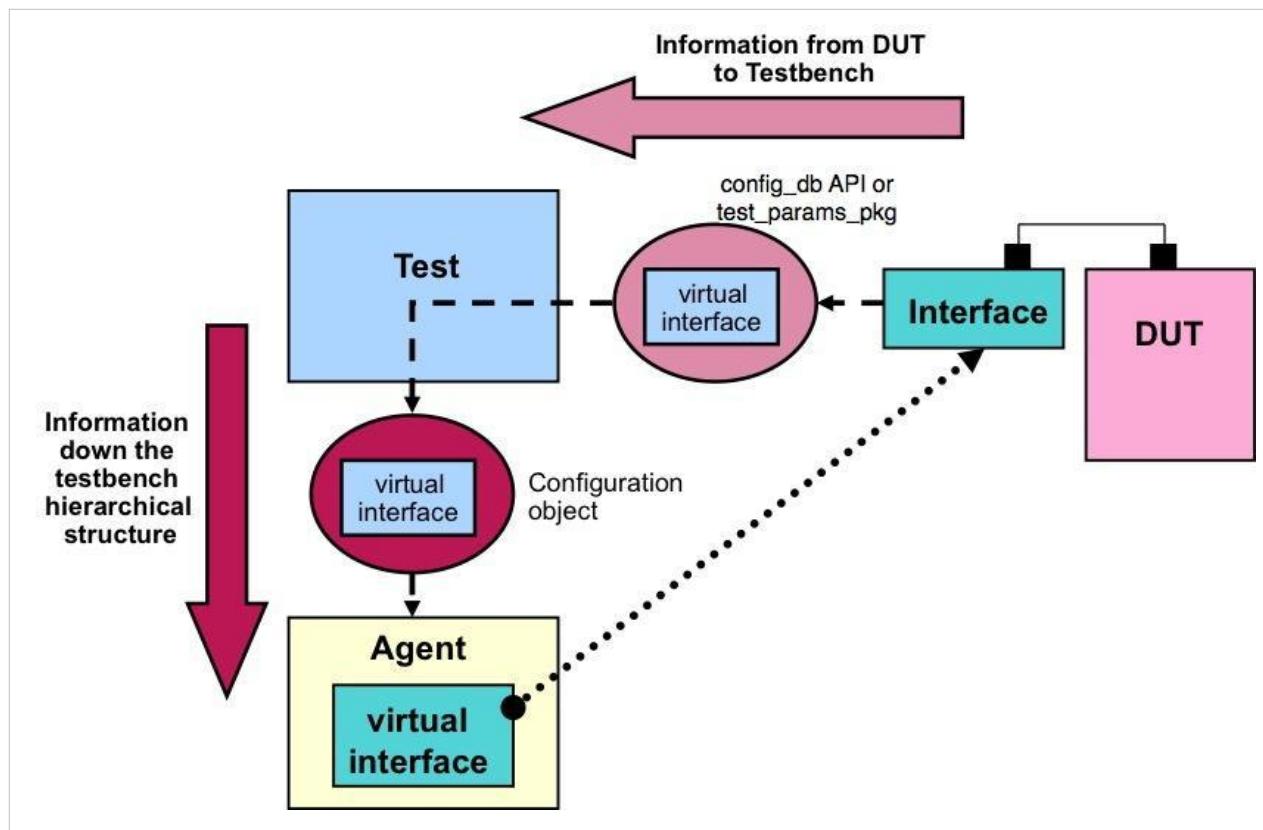
( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

## Setting Virtual Interface Properties in the Testbench

There are two separate and distinct activities that occur in order to set a virtual interface property in the transactors (driver, monitor etc.) of the testbench to point to the interface instance connected to the DUT.

First, the DUT provides connection information to the testbench. For the purposes of modularity and reuse this should be done in such a way that the DUT knows nothing about the testbench. The DUT should provide the necessary connection information to the testbench in a manner that does not require additional connections or modifications to the DUT or its associated logic.

Second, the connection information must be distributed in the testbench to the appropriate agents with their transactors. Setting and distribution of configuration and connection information in the testbench ideally should be done in a single place - the test class. The test class gets the information from the DUT and distributes it to the appropriate testbench components. The test class should be the only testbench component that has hard coded information about the DUT, and its interfaces. When DUT configuration changes occur then changes to the testbench can be limited to the test. This facilitates scalability and reusability for the rest of the testbench.



The questions may be asked: "Why not have the agents get the connection information directly from the DUT? Why have it distributed by the test? Doesn't that seem more complicated and extra work?"

The approach of the agents getting the information directly effectively hard codes information in the agents or transactors about the DUT and reduces scalability and reuse. If a change is made in the DUT configuration, it is likely that change would be required in the agent. One may think of the DUT connection and configuration information as a "pool" of information provided by the DUT to the testbench. In the recommended approach the test class gets information out of this pool and distributes it to the correct agents. If the information pool changes then appropriate changes are made in one location - the test. The agents are not affected because they get their information in the same manner - from the test. If instead the agents each get information directly from the pool they need to know which information to fetch. If the information pool changes then changes would need to be made in the agents.

There are two approaches to passing the location of the interface instance to the test class. The recommended approach is the first listed here which is using the config\_db API.

- Using config\_db API (recommended)
- Using a package

## Bus Functional Models (BFM)

Sometimes the DUT connection is not made directly to the ports of the DUT but rather is made through a BFM. Most often the BFM will have tasks for generating DUT transactions. DUT-TB communication with a BFM is discussed [here](#).

## Multiple Interface Instance Considerations

When you have multiple instances of an interface type, each instance needs to have some sort of unique ID so that the transactors may know which interface to be connected to. This may be done by simply appending a numeric value to the end of the instance name if the interface instances are declared in an array or at the same level. Often it is more convenient when there are multiple DUT instances and hence multiple interface instances to wrap the dut and its interfaces in a wrapper module. The wrapper module may be parameterized to form unique id's internal to the wrapper for multiple instances. In the wishbone wrapper module code shown above a WB\_ID parameter is used to uniquify the wishbone bus interface instance and facilitate the placement in the configuration database using uvm\_container. The code below shows the top level module with wrapper module instances:

```
module top_mac;

    wb_bus_wrapper #(0) wb_bus_0();
    wb_bus_wrapper #(1) wb_bus_1();

endmodule
```

In the test class the appropriate virtual interface is extracted and assigned to the appropriate wishbone configuration object. Wishbone environment 0 is then connected to wishbone bus wrapper 0 and so on.

```
class test_mac_simple_duplex extends uvm_test;
  ...
  mac_env env_0; //environment for WISHBONE bus 0
  mac_env env_1; //environment for WISHBONE bus 1
  wb_config wb_config_0; // config object for WISHBONE BUS 0
  wb_config wb_config_1; // config object for WISHBONE BUS 1
  ...

  function void set_wishbone_config_params();
    //set configuration info for WISHBONE 0
    wb_config_0 = new();
    wb_config_0.m_wb_id = 0; // WISHBONE 0
    // Get the virtual interface handle that was set in the top module or protocol module
    if (!uvm_config_db #(virtual wishbone_bus_bfm_if)::get(this, "", "WB_BFM_IF_0",
      wb_config_0.m_v_wb_bus_bfm_if)) // virtual interface
      `uvm_fatal("TEST_MAC_SIMPLE_DUPLEX", "Can't read WB_BFM_IF_0");
    uvm_config_db#(wb_config)::set(this, "env_0*", "wb_config", wb_config_0); // put in config
    ...
    wb_config_1 = new();
    wb_config_1.m_wb_id = 1; // WISHBONE 1
  endfunction
endclass
```

```

if (!uvm_config_db #(virtual wishbone_bus_bfm_if)::get(this, "",

                                         "WB_BFM_IF_1",
                                         wb_config_1.m_v_wb_bus_bfm_if)) // virtual interface

`uvm_fatal("TEST_MAC_SIMPLE_DUPLEX", "Can't read WB_BFM_IF_1");

uvm_config_db#(wb_config)::set(this, "env_1*", "wb_config", wb_config_1); // put in config

...
endfunction

...
endclass

```

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

If generate statements are used, the virtual interface assignment must also be done inside the generate block, because variable indexing of generate block instances is not allowed.

```

genvar i;

for (i = 0; i<NUM_INTERFACES; i++) begin : gen

  // alu_if instance
  alu_if a_if(.clk(clk));

  // DUT instance
  alu_rtl alu (
    .val1(a_if.val1),
    .val2(a_if.val2),
    .mode(a_if.mode),
    .clk(a_if.clk),
    .valid_i(a_if.valid_i),
    .valid_o(a_if.valid_o),
    .result(a_if.result)
  );
  initial begin
    // Each virtual interface must have a unique name, so use $sformatf
    uvm_config_db #(virtual alu_if)::set(null, "uvm_test_top",
                                         $sformatf("ALU_IF_%0d",i), a_if);
  end
end

```

# Config/VirtInterfaceConfigDb

## Setting Virtual Interface Properties in the Testbench with the Configuration Database using the config\_db API

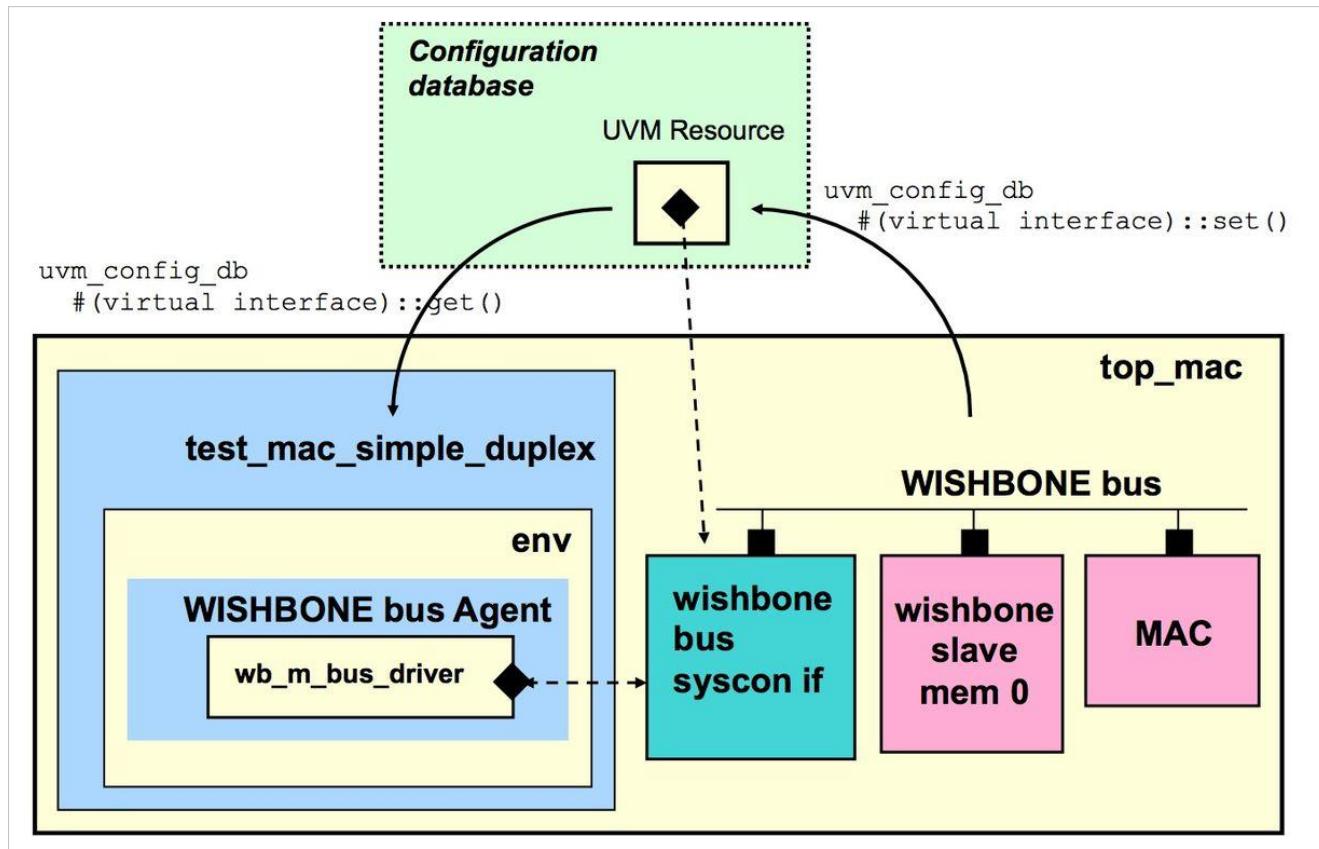
This is the **recommended approach** in assigning the actual interface reference to the virtual interface handles inside the testbench. This approach in general has three steps.

1. Use the config\_db API as a means to put a virtual interface, that points to a specific interface instance, into the configuration database.
2. The test class fetches the virtual interface from the configuration database and places it in a configuration object that is made available for the particular components (agents, drivers, monitors etc.) that communicate with the DUT through that particular interface.
3. The component that actually accesses the DUT via the virtual interface sets its virtual interface property from the virtual interface in the supplied configuration object.

There is a discussion here as to why one would take the approach of the test class fetching and distributing the information to the agents and transactors instead of having the transactors or agents fetch the data directly.

This approach supports scalability and reuse:

- Since the transactor receives the interface instance information from the configuration object, it is not affected by changes in the DUT configuration.
- If you are using emulation, this method works with protocol modules in the "Dual Top" methodology.



### Placing a Virtual Interface into the Configuration Database using the config\_db API

Unlike OVM where code was limited to only storing integral values, strings, or objects that derive from `ovm_object`, UVM allows full flexibility for all types of information to be stored in the resource/configuration database. This includes virtual interface handles, user defined types, etc. When the `config_db` API is used, a parameter is set with the type of information to be stored. This allows for simple `set()` and `get()` calls to be made without the need to wrap information in container objects such as the `ovm_container`.

```
// Top level module for a wishbone system with bus connection
// multiple masters and slaves
module top_mac;
...
// WISHBONE interface instance
// Supports up to 8 masters and up to 8 slaves
wishbone_bus_syscon_if wb_bus_if();
...

initial begin
  //set interfaces in config space
  uvm_config_db #(virtual wishbone_bus_syscon_if)::set(null, "uvm_test_top",
                                                       "WB_BUS_IF", wb_bus_if);

  run_test("test_mac_simple_duplex"); // create and start running test
end

endmodule
```

Notes:

- We use "`uvm_test_top`" because it is more precise and more efficient than "\*". This will always work unless your top level test does something other than `super.new( name , parent )` in the constructor of your test component. If you do something different in your test component, you will have to adjust the instance name accordingly.
- We use "`null`" in the first argument because this code is in a top level module, not a component.

### Making the Virtual Interface Available in the Testbench

The test class creates a configuration object which has a virtual interface property. It then assigns this property by calling the `config_db::get()` function which delivers the virtual interface from the configuration database. The second argument to this method must be the same string that was used to place the virtual interface in the configuration database.

The test then places the configuration object into the configuration database to provide access for the particular components (agents, drivers, monitors etc.) that communicate with the DUT through that particular interface.

```
class test_mac_simple_duplex extends uvm_test;
...
wb_config wb_config_0; // config object for WISHBONE BUS
...
```

```

function void set_wishbone_config_params();
  //set configuration info
  wb_config_0 = new();

  if (!uvm_config_db #(virtual wishbone_bus_syscon_if)::get(this, "", "WB_BUS_IF",
    wb_config_0.v_wb_bus_if)) // virtual interface
    `uvm_fatal("TEST_MAC_SIMPLE_DUPLEX", "Can't read WB_BUS_IF");

  ... // other WISHBONE bus configuration settings

  uvm_config_db#(wb_config)::set(this, "*", "wb_config", wb_config_0); // put in config

endfunction

function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  set_wishbone_config_params();
  ...
endfunction : build_phase
...
endclass

```

### Assigning Virtual Interface Property in Transactor

The component that actually accesses the DUT via the virtual interface sets its virtual interface property from the virtual interface in the supplied configuration object.

```

// WISHBONE master driver
class wb_m_bus_driver extends uvm_driver #(wb_txn, wb_txn);
  ...
  virtual wishbone_bus_syscon_if m_v_wb_bus_if; // Virtual Interface
  wb_config m_config;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db #(wb_config)::get(this, "", "wb_config", m_config)) // get config object
      `uvm_fatal("Config Fatal", "Can't get the wb_config")
    ...
  endfunction : build_phase

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    m_v_wb_bus_if = m_config.v_wb_bus_if; // set local virtual if property
  endfunction : connect_phase

```

```
 ...  
endclass
```

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

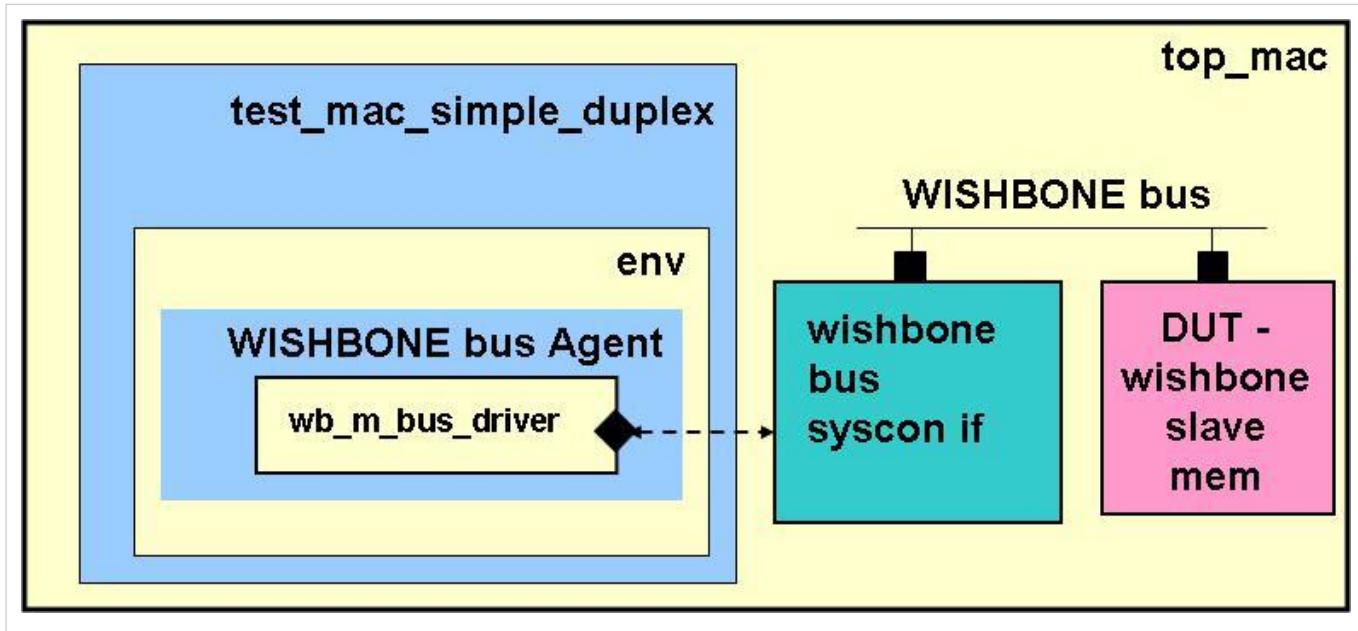
# Connect/VirtInterfacePackage

## Setting Virtual Interface Properties in the Testbench with Packages

An easy way of assigning the actual interface reference to the virtual interface handles inside the testbench is by creating a virtual interface variable in a package. This method has the advantage of simplicity. However, because of its disadvantages this approach should only be considered for relative simple designs that do not have parameterized interfaces, or do not have multiple instances of an interface and is **not recommended for general use**.

It has the following disadvantages that limit reusability:

- Parameterized interfaces cannot be declared in the package with generic parameter values - they must use actual values. Any changes to parameter values would then force a recompilation.
- It introduces an additional dependency on an external variable. So, for example, any change to the virtual interface name would require changes in any components that referenced the variable.



## Virtual Interface Variable in Package

In this connection method, the virtual interface variable is passed to the testbench test via a package. In this example, the package used is the agent package, but it could be any package shared between the top level module and the UVM test package.

```
package wishbone_pkg;
...
virtual wishbone_bus_syscon_if v_wb_bus_if; // virtual wishbone interface pointer
...
`include "wb_m_bus_driver.svh"
`include "wb_bus_monitor.svh"
...
```

```
endpackage
```

In the top level module, just assign the actual interface instance to the package variable:

```
module top_mac;
...
// WISHBONE interface instance
wishbone_bus_syscon_if wb_bus_if();
...

initial begin
  //set virtual interface to wishbone bus
  wishbone_pkg::v_wb_bus_if = wb_bus_if;
  ...

end
endmodule
```

Then in the test, the virtual interface handle variable in the package should be assigned to a virtual interface handle in a component (agent) configuration object.

```
class test_mac_simple_duplex extends uvm_test;

wb_config wb_config_0; // config object for WISHBONE BUS

function void build_phase(uvm_phase phase);
  wb_config_0 = wb_config::type_id::create("wb_config_0");
  // ...
  wb_config_0.v_wb_bus_if = wishbone_pkg::v_wb_bus_if; // From the wishbone_pkg
  uvm_config_db #(wb_config)::set(this, "*", "wb_config", wb_config_0);
  // ...
endfunction: build_phase
```

Any component that uses the virtual interface should create a local handle and assign the configuration object handle to the local handle in the connect() method.

```
// wishbone master driver
class wb_m_bus_driver extends uvm_driver #(wb_txn, wb_txn);
...

virtual wishbone_bus_syscon_if m_v_wb_bus_if;
wb_config m_config;
...

function void build_phase(uvm_phase phase);
  // get config object
```

```
if (!uvm_config_db#(wb_config)::get(this,"","wb_config", m_config) )
  `uvm_fatal("CONFIG_LOAD", "Cannot get() configuration wb_config from uvm_config_db. Have you set() it?")
  m_id = m_config.m_wb_master_id;
endfunction

function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);
  m_v_wb_bus_if = m_config.v_wb_bus_if; // set local virtual if property
endfunction : connect_phase
...

endclass
```

Note that using the agent package to share the virtual interface handle has a severe impact - only one instance of the agent can be used in the UVM testbench, since there is only virtual interface handle available.

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# Connect/VirtInterfaceConfigPkg

---

## Setting Virtual Interface Properties in the Tesbench using a Package

An alternative to using the config\_db API to provide virtual interface information to the test class is to use a package. The recommended approach is to use the test parameters package. An alternate approach, which is not recommended but is in industry use is to use the agent package. This article will focus on the recommended approach.

### Using the Test Parameters Package

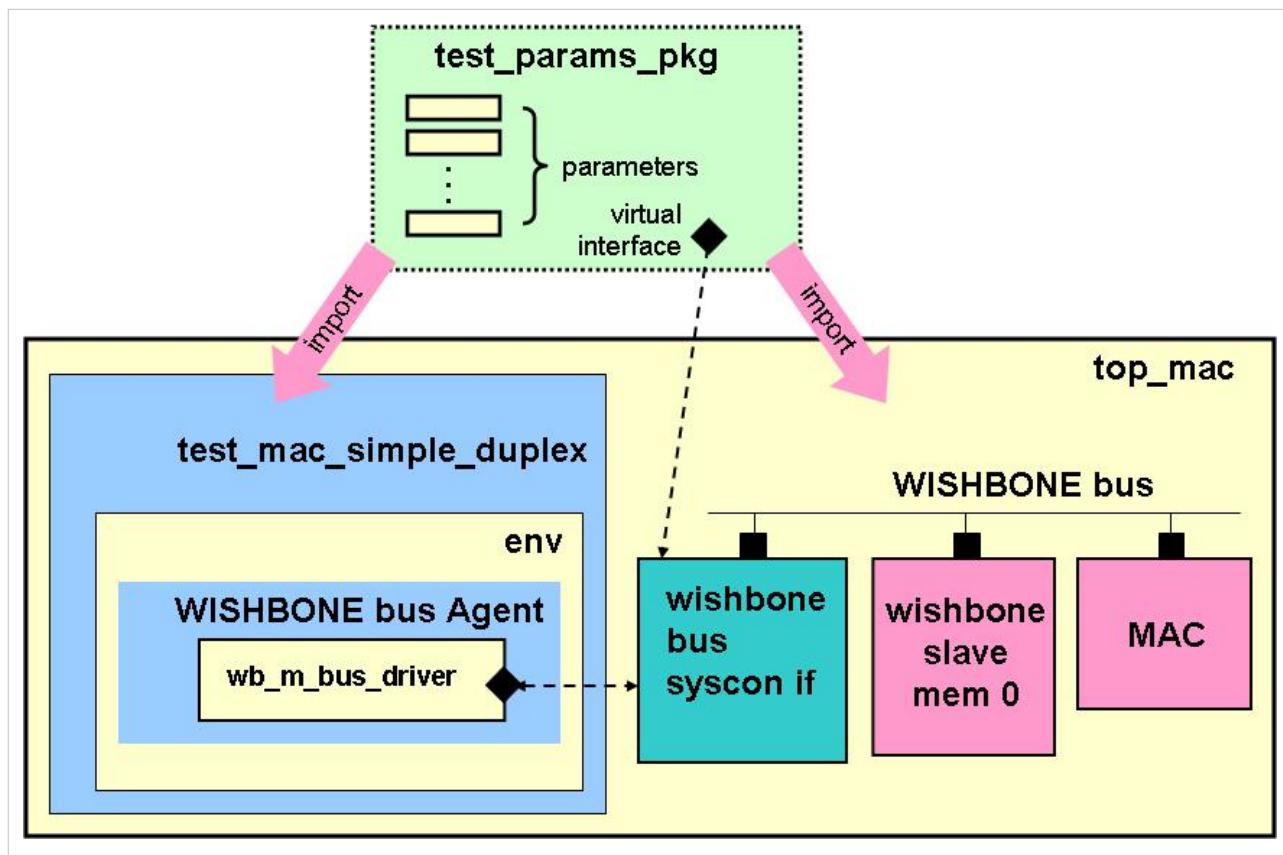
This approach to assigning the actual interface reference to the virtual interface handles inside the testbench is very similar to the recommended approach using the config\_db API. It is the same in that the test uses a configuration object and the configuration database to distribute the virtual interface. It differs in that a package is used to convey the virtual interface handle to the test class instead of using the config\_db API.

The disadvantage of this approach is it can not be used with protocol modules which means this approach can not easily support emulation.

This approach in general has 4 steps.

1. A virtual interface is declared in a package. It is recommended that this package be the test parameters package.
2. In the top module containing the DUT and interface instance an assignment is made to the virtual interface in the package to point to the interface instance.
3. Test class gets the virtual interface information from the package and assigns the virtual interface in a configuration object that is distributed to the appropriate components (agents, drivers, monitors etc.) that communicate with the DUT through that particular interface.
4. The component that actually accesses the DUT via the virtual interface sets its virtual interface property from the virtual interface in the supplied configuration object.

There is a discussion here as to why one would take the approach of the test class fetching and distributing the information to the agents and transactors instead of having the transactors or agents fetch the data directly.



### Declaring Virtual Interface in the Test Parameters Package

In the test parameters package declare a virtual interface pointer.

```
package test_params_pkg;
import uvm_pkg::*;

// WISHBONE bus virtual interface
virtual wishbone_bus_syscon_if v_wb_bus_if;

...
endpackage
```

### Assign Virtual Interface in the Test Parameters Package

In the top module containing the DUT and the interface instance make an assignment in an initial block to point the virtual interface in the test parameters package to the interface instance. The virtual interface property may be imported or explicitly referenced.

```
module top_mac;
...
import test_params_pkg::*;


```

```

// WISHBONE interface instance
// Supports up to 8 masters and up to 8 slaves
wishbone_bus_syscon_if wb_bus_if();

...

initial begin
  //set WISHBONE virtual interface in test_params_pkg
  v_wb_bus_if = wb_bus_if;

  run_test("test_mac_simple_duplex"); // create and start running test
end

endmodule

```

### Making the Virtual Interface available in the Testbench

The test class creates a configuration object which has a virtual interface property. It then assigns this property from the virtual interface in the test parameters package.

The test then places the configuration object into the configuration database for providing access for the particular components (agents, drivers, monitors etc.) that communicate with the DUT through that particular interface.

```

class test_mac_simple_duplex extends uvm_test;
  ...

  wb_config wb_config_0; // config object for WISHBONE BUS
  ...

  function void set_wishbone_config_params();
    //set configuration info
    wb_config_0 = new();

    // Set WISHBONE bus virtual interface in config obj to virtual interface in test_params_pkg
    wb_config_0.v_wb_bus_if = v_wb_bus_if;
    ...

    uvm_config_db#(wb_config)::set(this, "*", "wb_config", wb_config_0); // put in config
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_wishbone_config_params();
    ...
  endfunction : build_phase
  ...
endclass

```

### Setting Virtual Interface Property in Transactor

The component that actually accesses the DUT via the virtual interface sets its virtual interface property from the virtual interface property in the supplied configuration object.

```
// WISHBONE master driver
class wb_m_bus_driver extends uvm_driver #(wb_txn, wb_txn);
  ...
  virtual wishbone_bus_syscon_if m_v_wb_bus_if; // Virtual Interface
  wb_config m_config;
  ...
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db #(wb_config)::get(this, "", "wb_config", m_config)) // get config object
      `uvm_fatal("Config Fatal", "Can't get the wb_config")
    ...
  endfunction : build_phase

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    m_v_wb_bus_if = m_config.v_wb_bus_if; // set local virtual if property
  endfunction : connect_phase
  ...
endclass
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Connect/TwoKingdomsFactory

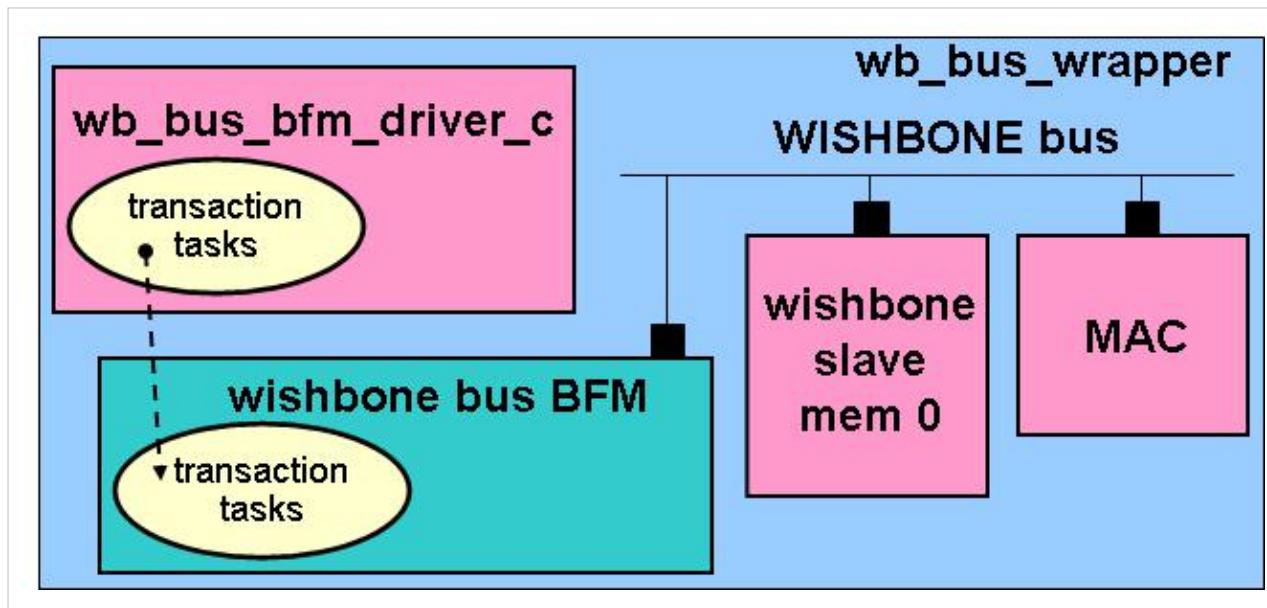
## Abstract/base Class

The Abstract/base class is defined as part of the agent in the testbench. In this example it is a base class driver and includes ports for connection to the rest of the wishbone bus agent.

## Wrapper module

A wrapper module is created that includes instances of the BFM and the DUT. The concrete class is defined inside the module so its scope will be the wrapper module.

An instance of the concrete class is created inside of the wrapper module. A handle to this instance is placed inside the configuration database using uvm\_container.



Below is the code for the WISHBONE bus BFM.

```

module wishbone_bus_syscon_bfm #(int num_masters = 8, int num_slaves = 8,
                                int data_width = 32, int addr_width = 32) ();
    ...
    // WISHBONE signals
    bit clk;
    bit rst;
    ...
    // WISHBONE bus arbitration logic
    ...
    // Slave address decode
    ...
    // BFM tasks
    //WRITE 1 or more write cycles
    task wb_write_cycle(wb_txn req_txn, bit [2:0] m_id = 1);

```

```

...
endtask

//READ 1 or more cycles
task wb_read_cycle(wb_txn req_txn, bit [2:0] m_id = 1, output wb_txn rsp_txn);
...
endtask

// Monitor bus transactions
task monitor(output wb_txn txn);
...
endtask

endmodule

```

Below is the code for the WISHBONE bus wrapper module. Note the instances of the BFM (wishbone\_bus\_syscon\_bfm), the slave memory (wb\_slave\_mem) and the Ethernet MAC (eth\_top). The MAC chip also has a Media Independent Interface (MII) besides the WISHBONE interface that is not shown or discussed. There are actually two concrete classes defined in this bus wrapper - a driver and a monitor but only the driver (wb\_bus\_bfm\_driver\_c) is shown and discussed.

```

module wb_bus_wrapper #(int WB_ID = 0);
...
// WISHBONE BFM instance
// Supports up to 8 masters and up to 8 slaves
wishbone_bus_syscon_bfm wb_bfm();

// WISHBONE 0, slave 0: 000000 - 0fffff
wb_slave_mem #(mem_slave_size) wb_s_0 ( ... );

// MAC 0
// It is WISHBONE slave 1: address range 100000 - 100fff
// It is WISHBONE Master 0
eth_top mac_0 ( ... );

// Concrete driver class
class wb_bus_bfm_driver_c #(int ID = WB_ID) extends wb_bus_bfm_driver_base;
...
endclass
...
endmodule

```

If for example the concrete driver class receives a WISHBONE write transaction it calls its local wb\_write\_cycle() task which in turn calls the wb\_write\_cycle() task inside the BFM.

```

module wb_bus_wrapper #(int WB_ID = 0);
  ...

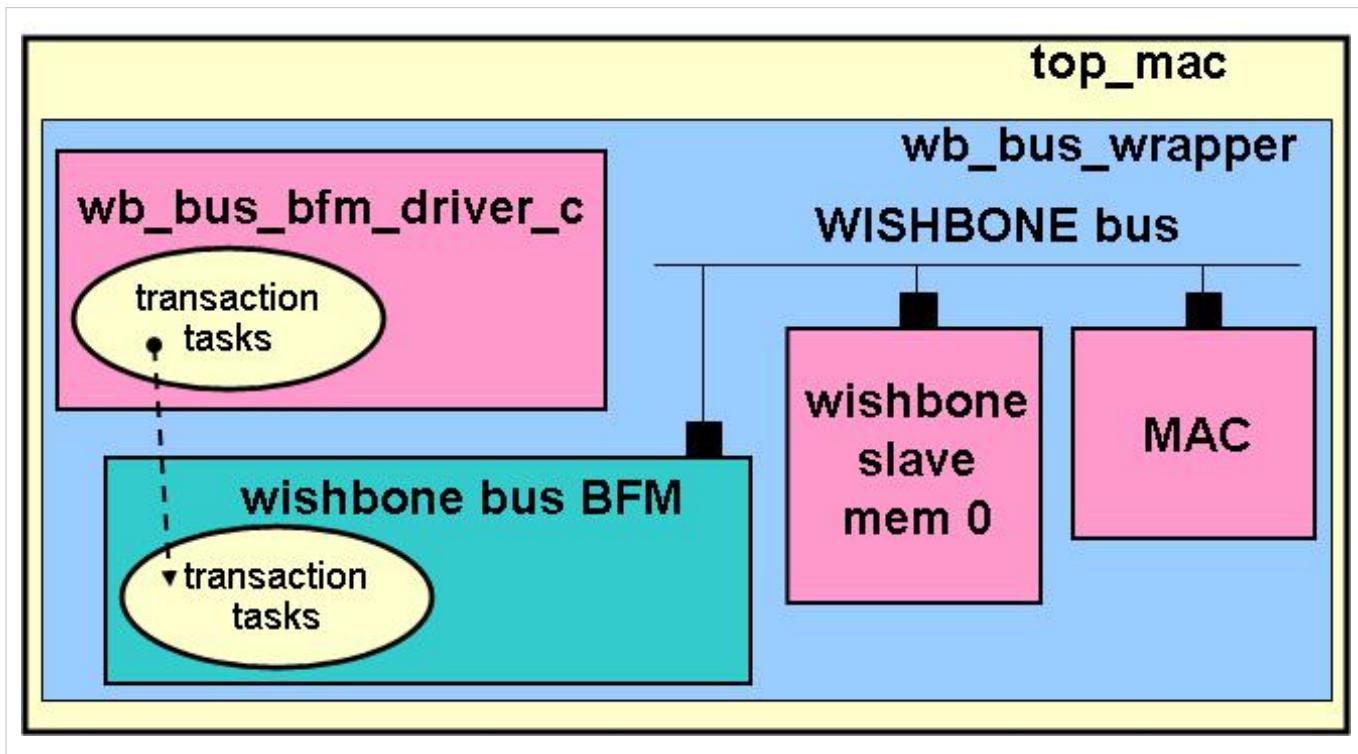
  // Concrete driver class
  class wb_bus_bfm_driver_c #(int ID = WB_ID) extends wb_bus_bfm_driver_base;
    ...

    task run_phase(uvm_phase phase);
      wb_txn req_txn;
      forever begin
        seq_item_port.get(req_txn); // get transaction
        @ (posedge wb_bfm.clk) #1; // sync to clock edge + 1 time step
        case(req_txn.txn_type) //what type of transaction?
          NONE: `uvm_info($sformatf("WB_M_DRVR_%0d",m_id),
                        $sformatf("wb_txn %0d the wb_txn_type was type NONE",
                        req_txn.get_transaction_id()),UVM_LOW )
          WRITE: wb_write_cycle(req_txn);
          READ: wb_read_cycle(req_txn);
          RMW: wb_rmw_cycle(req_txn);
          WAIT_IRQ: fork wb_irq(req_txn); join_none
          default: `uvm_error($sformatf("WB_M_DRVR_%0d",m_id),
                        $sformatf("wb_txn %0d the wb_txn_type was type illegal",
                        req_txn.get_transaction_id()) )
        endcase
      end
    endtask : run_phase

    // Methods
    // calls corresponding BFM methods
    //WRITE 1 or more write cycles
    task wb_write_cycle(wb_txn req_txn);
      wb_txn orig_req_txn;
      $cast(orig_req_txn, req_txn.clone()); //save off copy of original req transaction
      wb_bfm.wb_write_cycle(req_txn, m_id);
      wb_drv_ap.write(orig_req_txn); //broadcast original transaction
    endtask
    ...
  endclass
  ...
endmodule

```

## Connecting the testbench to the DUT



In the wishbone wrapper an instance override is created of the derived concrete class driver for the base class driver. Note the code in this example is set up to handle multiple instances of the wishbone bus wrapper (and hence multiple DUTs) which is the reason for the use of the WB\_ID parameters to uniquify the instances override. This parameter is set in the top\_mac module.

```
module top_mac;

  wb_bus_wrapper #(0) wb_bus_0();

  ...

endmodule

module wb_bus_wrapper #(int WB_ID = 0);
  ...

  // Concrete driver class
  class wb_bus_bfm_driver_c #(int ID = WB_ID) extends wb_bus_bfm_driver_base;
    ...
  endclass

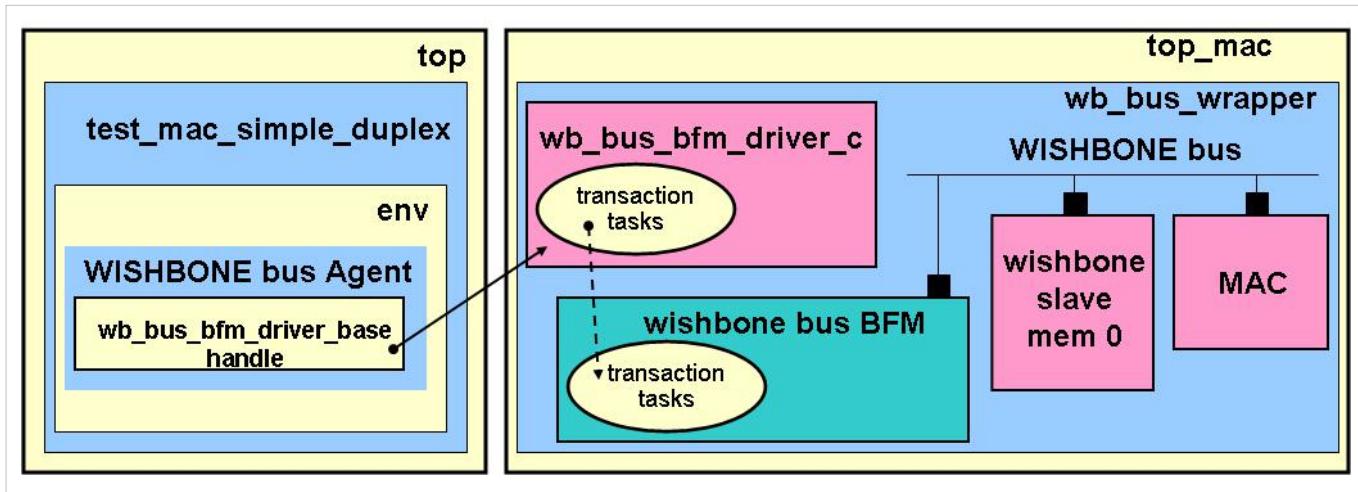
  initial begin
    //set inst override of concrete bfm driver for base bfm driver
  end
endmodule
```

```

wb_bus_bfm_driver_base::type_id::set_inst_override(
    wb_bus_bfm_driver_c #(WB_ID)::get_type(),
    $sformatf("*env_%0d*", WB_ID));
...
end

endmodule

```



In the wishbone agent in the testbench a base class driver handle is declared. When it is created by the factory the override will take effect and a derived concrete class driver object will be created instead. This object has as its Verilog scope the wishbone bus wrapper and its Verilog path would indicate it is inside the wishbone bus wrapper instance created inside of top\_mac (wb\_bus\_0). From the UVM perspective the object is an UVM hierarchical child of the wishbone agent and its ports are connected to the ports in the wishbone agent.

```

class wb_master_agent extends uvm_agent;
...
//ports
uvm_analysis_port #(wb_txn) wb_agent_drv_ap;
...
// components
wb_bus_bfm_driver_base wb_drv;
wb_config m_config;
...
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db #(wb_config)::get(this, "", "wb_config", m_config)) // get config object
        `uvm_fatal("Config Fatal", "Can't get the wb_config")
    //ports
    wb_agent_drv_ap = new("wb_agent_drv_ap", this);
    ...
//components
wb_drv = wb_bus_bfm_driver_base::type_id::create("wb_drv", this); // driver

```

```
...
endfunction : build_phase

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    //analysis ports
    wb_drv.wb_drv_ap.connect(wb_agent_drv_ap);
    ...
    // child ports
    wb_drv.seq_item_port.connect(wb_seqr.seq_item_export);
endfunction : connect_phase

function void end_of_elaboration_phase(uvm_phase phase);
    wb_drv.m_id = m_config.m_wb_master_id;
    ...
endfunction : end_of_elaboration_phase
endclass
```

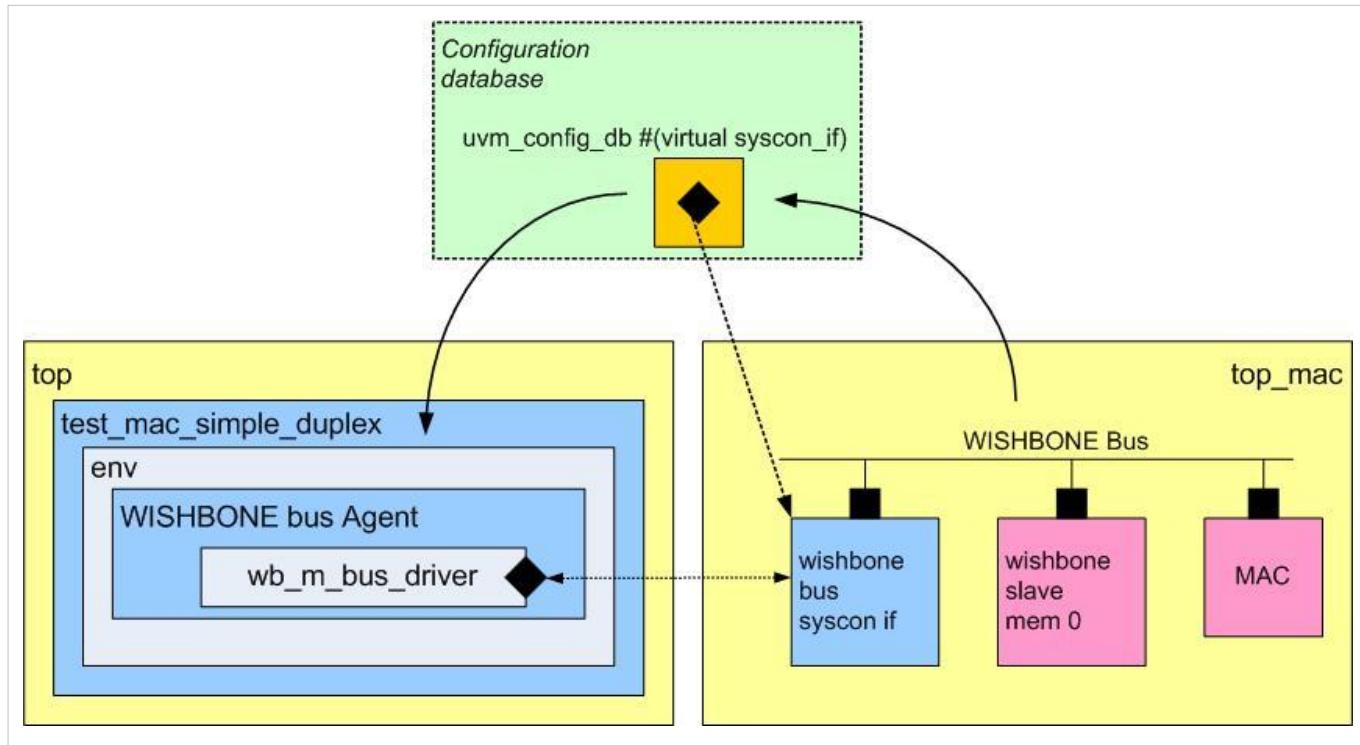
( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# DualTop

Typically a DUT-TB setup has a single SystemVerilog module as the top level. This top level module contains the DUT and its associated interfaces, protocol modules, connection and support logic. It also contains the code to create the testbench. All this "stuff" can get messy and hard to manage. A different way to manage all this stuff is to encapsulate everything associated directly with the DUT in a wrapper module. The code to create the testbench is placed in its own module. Verilog allows for more than one top level module in a simulation. Neither of these two modules are instantiated but rather are treated as top level modules. This arrangement is referred to as dual top.

Dual top is a necessity for emulation. The DUT wrapper is the stuff that goes in the emulator. The other top module containing the testbench stays running in the simulator. If the testbench is only going to be used in simulation dual top is not necessary but may however still provide a useful level of encapsulation for modularity, reuse etc.

Communicating the virtual interface connection between the DUT wrapper module and the testbench is done using the `uvm_config_db`.



## DUT Wrapper module

In this example the MAC DUT and its associated interfaces and protocol module and the WISHBONE slave memory along with the WISHBONE bus logic are contained in DUT wrapper module `top_mac`

```
module top_mac;
  import uvm_pkg::*;
  import test_params_pkg::*;

  // WISHBONE interface instance
  // Supports up to 8 masters and up to 8 slaves
```

```

wishbone_bus_syscon_if wb_bus_if();

//-----
// WISHBONE 0, slave 0: 000000 - 0fffff
// this is 1 Mbytes of memory
wb_slave_mem #(mem_slave_size) wb_s_0 (
  ...
);

// wires for MAC MII connection
wire [3:0] MTxD;
wire [3:0] MRxD;

//-----
// MAC 0
// It is WISHBONE slave 1: address range 100000 - 100fff
// It is WISHBONE Master 0
eth_top mac_0 (
  ...
);

// protocol module for MAC MII interface
mac_mii_protocol_module #(.INTERFACE_NAME("MIIM_IF")) mii_pm(
  ...
);

initial
  //set WISHBONE virtual interface in config space
  uvm_config_db #(virtual wishbone_bus_syscon_if)::set(null, "uvm_test_top", "WB_BUS_IF", wb_bus_if);

endmodule

```

## Testbench top module

In the top module that creates the testbench is an initial block that calls `run_test()`. Note the imports `uvm_pkg::run_test` and `tests_pkg::test_mac_simple_duplex`. These are necessary for compilation of the `run_test()` line of code.

```

// This the top for the tesbench in a dual top setup
// The dut is in the top_mac dut wrapper module
module top;
  import uvm_pkg::run_test;
  import tests_pkg::test_mac_simple_duplex;

  initial
    run_test("test_mac_simple_duplex"); // create and start running test

```

```
endmodule
```

## Questasim with more than one top level module

The call to vsim includes more than one top module

```
#Makefile
...
normal: clean cmp
    vsim +nowarnTSCALE -c top top_mac -suppress 3829 -do "run -all"
...
```

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# VirtInterfaceFunctionCallChain

---

## Function Call Chaining

It is unfortunate that this approach to assigning actual interface reference to the virtual interface handles inside the testbench is the one that is used in the xbus example that is prevalent in the UVM user's guide. Many users naturally assume that this is the recommended method because of its use in the example. **This approach is not recommended.**

It involves creating a function (called assign\_vi in the example) that takes a virtual interface handle as an argument, and calls an equivalent function (also named assign\_vi) on one or more child components. This is repeated down the hierarchy until a leaf component is reached. Any components that need the virtual interface declare a local handle and assign the function argument to the local handle.

In the connect() function of test env:

```
xbus0.assign_vi(xbus_tb_top.xi0);
```

In the xbus env:

```
function void assign_vi(virtual interface xbus_if xi);
    xi0 = xi;
    if( bus_monitor != null) begin
        bus_monitor.assign_vi(xi);
    end
    for(int i = 0; i < num_masters; i++) begin
        masters[i].assign_vi(xi);
    end
    for(int i = 0; i < num_slaves; i++) begin
        slaves[i].assign_vi(xi);
    end
endfunction : assign_vi
```

In the agent:

```
function void assign_vi(virtual interface xbus_if xmi);
    monitor.assign_vi(xmi);
    if (is_active == UVM_ACTIVE) begin
        sequencer.assign_vi(xmi);
        driver.assign_vi(xmi);
    end
endfunction : assign_vi
```

In the monitor:

```
function void assign_vi(virtual interface xbus_if xmi);
    this.xmi = xmi;
endfunction
```

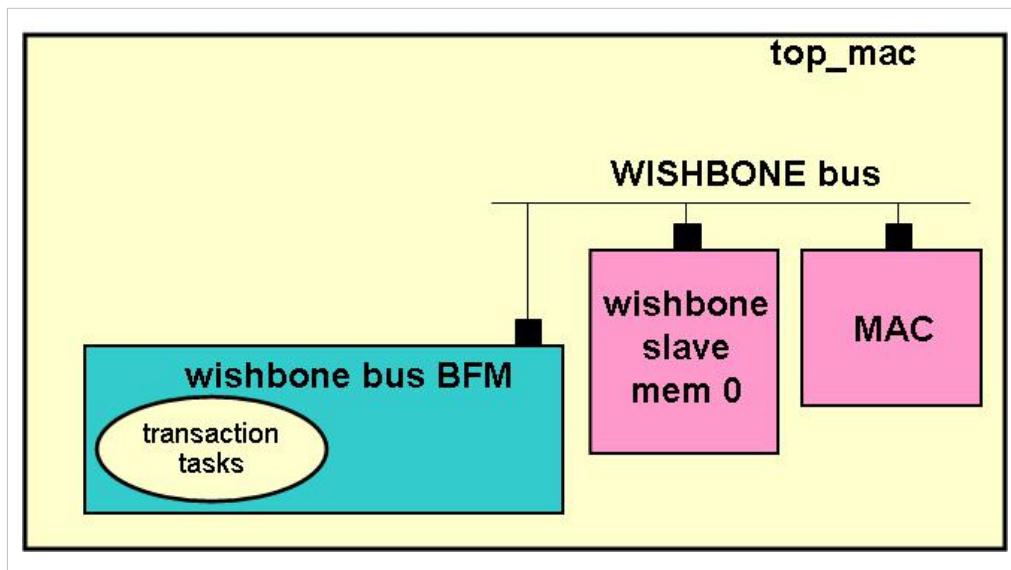
There are two main reasons why this method should not be used.

- **It is not reusable** - If the test environment hierarchy changes, these functions must be updated
- **Unnecessary extra work** - To reach leaf components in the environment, you must pass the virtual interface handle down through intermediate levels of the hierarchy that have no use for the virtual interface. Also, to make this method more reusable with respect to environment hierarchy changes, you would have to embed extra decision-making code (as in the examples above). or write each function to iterate over all children or and call the function on each child. This requires even more unnecessary work.

# BusFunctionalModels

## Bus Functional Models for DUT communication

Sometimes a the DUT connection is not directly to the ports of the DUT but rather is made through a BFM. As shown in the diagram below, typically the BFM will have tasks for generating DUT transactions.



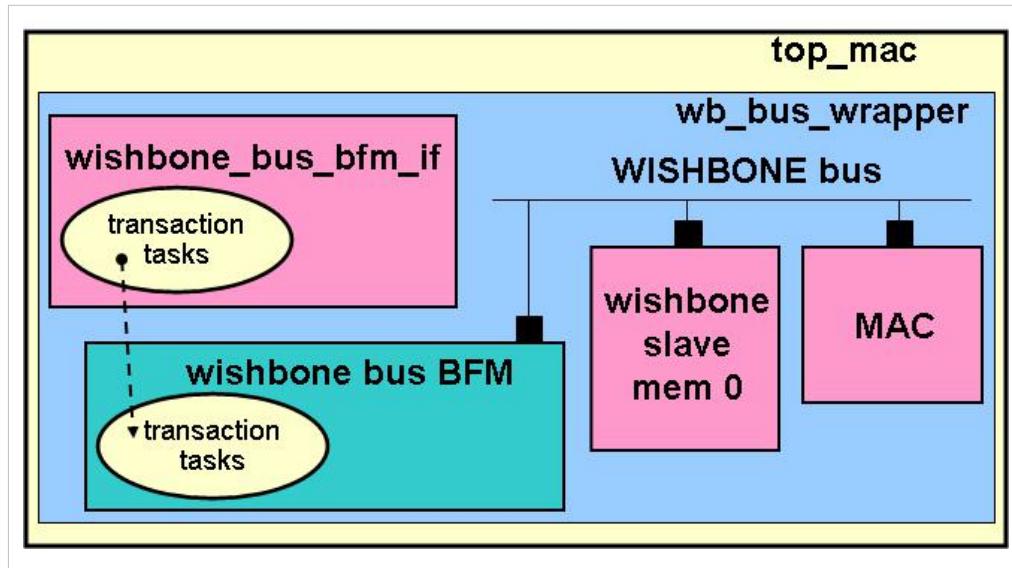
There are several different "flavors" of BFM:

### Verilog Module BFM - Can be modified

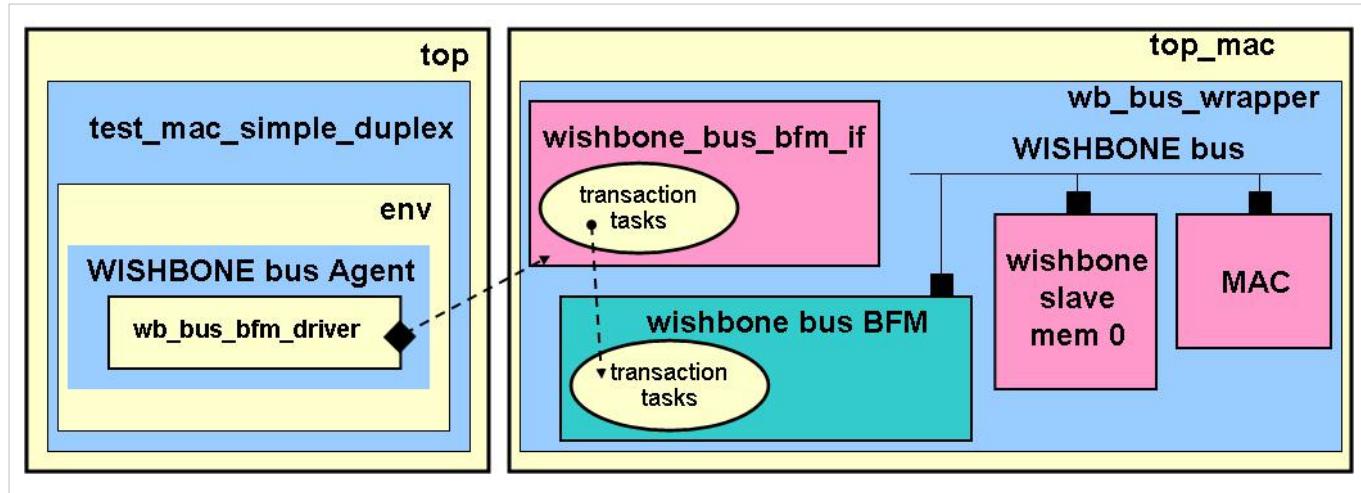
If the BFM is a Verilog module the first choice with this type of connection is to modify the BFM to be an interface. Then a standard *virtual interface connection* can be made to the BFM and the testbench transactors can directly access the BFM transaction tasks.

### Verilog Module BFM - Can not be modified

The Verilog BFM may not be able to be modified for several reasons. It may have an instance of a module in which case it can not be changed to an interface. Or it simply can not be edited. In this case a wrapper module is introduced. The wrapper module contains the instance of the BFM, the DUT and an instance of an interface as shown in the diagram below.



The interface acts as a "proxy" for the BFM to the testbench transactors. For example to do a wishbone bus write transaction the driver would call the write task in the interface which in turn would call the write task in the BFM. See the diagram below.



Example code for the wrapper module is below:

```
module wb_bus_wrapper #(int WB_ID = 0);
...
// WISHBONE BFM instance
wishbone_bus_syscon_bfm wb_bfm();

// WISHBONE 0, slave 0: 000000 - 0fffff
wb_slave_mem #(mem_slave_size) wb_s_0 (...);

// MAC 0
eth_top mac_0(...);
```

```
// Interface
interface wishbone_bus_bfm_if #(int ID = WB_ID)
  (input bit clk);
  // Methods
  //WRITE 1 or more write cycles
  task wb_write_cycle(wb_txn req_txn, bit [2:0] m_id);
    wb_bfm.wb_write_cycle(req_txn, m_id);
  endtask
  // other tasks not shown
  ...
endinterface

// Interface instance
wishbone_bus_bfm_if #(WB_ID) wb_bus_bfm_if(.clk(wb_bfm.clk));

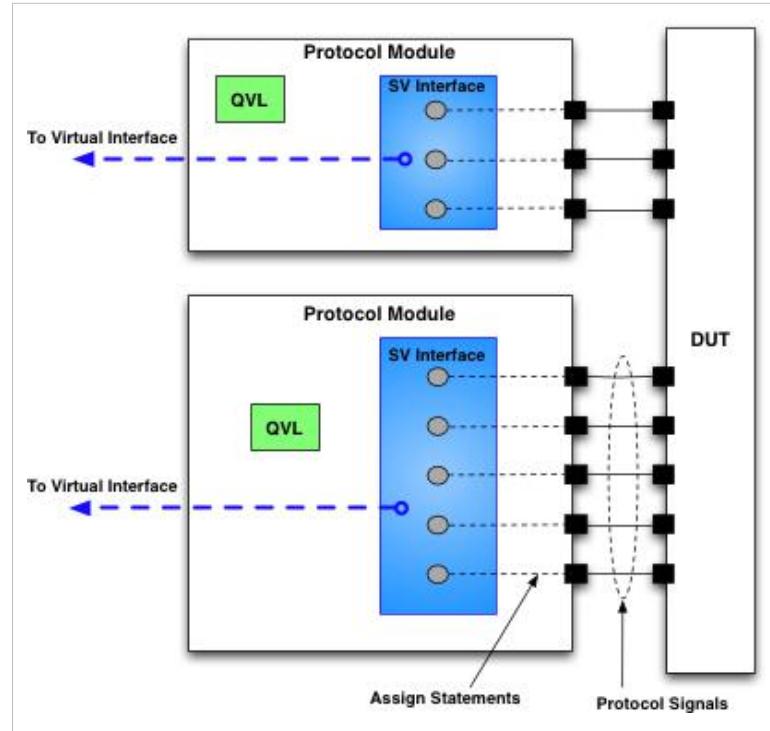
initial
  //set interface in config space
  uvm_config_db #(virtual wishbone_bus_bfm_if)::set(null, "uvm_test_top",
    $sformatf("WB_BFM_IF_%0d",WB_ID), wb_bus_bfm_if);
endmodule
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# ProtocolModules

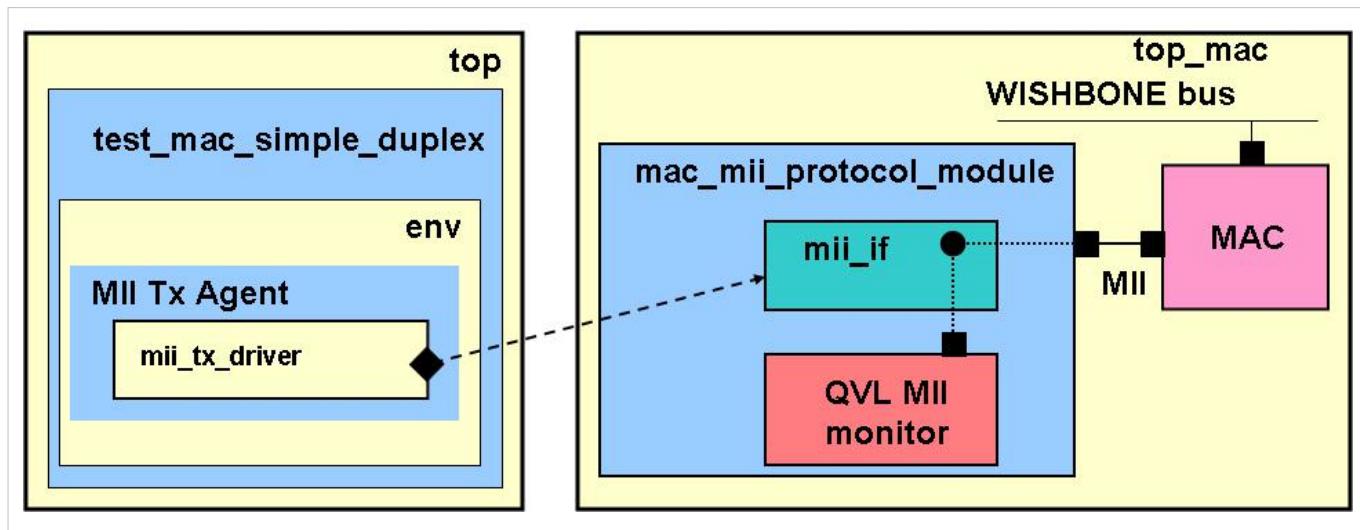
Protocol modules are wrapper modules that encapsulate a DUT interface, associated assertions, QVL instances (which are not allowed inside an interface), and so forth.

When emulation is a consideration protocol modules provide a level of encapsulation necessary to isolate the changes that occur in the agent and interface in moving between simulation and emulation. If the testbench is only going to be used in simulation protocol modules are not necessary. They may however still provide a useful level of encapsulation for modularity, reuse etc. While it is not required that protocol modules be used together with the ***dual top*** methodology it is likely to be used mainly in connection with the ***dual top*** approach since it is also required for emulation.



By adopting encapsulation, protocol modules protect the top level from changes:

- Any re-mapping due to changes in the interface can be done inside the protocol module.
- The top level module is protected from changes to the virtual interface registration/connection technique.
- You can instantiate QVL instances (which would not be allowed inside the SV interface) as well as add other assertions that might not already be present in the interface.

**Example:**

In this example an Ethernet Media Access Controller (MAC) is the DUT. A MAC has multiple interfaces. The one shown in the example is the Media Independent Interface (MII) which is where Ethernet packets are transferred to the physical interface. In this example the protocol module contains the MII interface instance, a QVL MII monitor and code for putting the interface instance location in the configuration database using UVM Resources.

```

module mac_mii_protocol_module #(string INTERFACE_NAME = "") (
    input    logic      wb_rst_i,
    // Tx
    output   logic      mtx_clk_pad_o, // Transmit clock (from PHY)
    input    logic[3:0]  mtdx_pad_o,   // Transmit nibble (to PHY)
    input    logic      mtxen_pad_o,  // Transmit enable (to PHY)
    input    logic      mtxerr_pad_o, // Transmit error (to PHY)

    // Rx
    output   logic      mrx_clk_pad_o, // Receive clock (from PHY)
    output   logic[3:0]  mrdx_pad_i,   // Receive nibble (from PHY)
    output   logic      mrdv_pad_i,   // Receive data valid (from PHY)
    output   logic      mrerr_pad_i,  // Receive data error (from PHY)

    // Common Tx and Rx
    output   logic      mcoll_pad_i,   // Collision (from PHY)
    output   logic      mcrs_pad_i,   // Carrier sense (from PHY)

    // MII Management interface
    output   logic      md_pad_i,     // MII data input (from I/O cell)
    input    logic      mdc_pad_o,    // MII Management data clock (to PHY)
    input    logic      md_pad_o,     // MII data output (to I/O cell)
    input    logic      md_padoe_o,   // MII data output enable (to I/O cell)
);

```

```
import uvm_container_pkg::*;

// Instantiate interface
mii_if miim_if();

// Connect interface to protocol signals through module ports
assign mtx_clk_pad_o = miim_if.mtx_clk;
assign miim_if.MTxD = mtdx_pad_o;
assign miim_if.MTxEn = mtxen_pad_o;
assign miim_if.MTxErr = mtxerr_pad_o;

assign mrx_clk_pad_o = miim_if.mrx_clk;
assign mrxd_pad_i = miim_if.MRxD;
assign mrxdv_pad_i = miim_if.MRxDV ;
assign mrxerr_pad_i = miim_if.MRxErr ;

assign mcoll_pad_i = miim_if.MColl    ;
assign mcrs_pad_i = miim_if.MCrs     ;

assign md_pad_i = miim_if.Mdi_I      ;
assign miim_if.Mdc_O = mdc_pad_o    ;
assign miim_if.Mdo_O = md_pad_o    ;
assign miim_if.Mdo_OE = md_padoe_o  ;

// Instantiate QVL Checker
qvl_gigabit_ethernet_mii_monitor mii_monitor(
    .areset(1'b0),
    .reset(wb_rst_i),
    .tx_clk(miim_if.mtx_clk),
    .txd(miim_if.MTxD),
    .tx_en(miim_if.MTxEn),
    .tx_er(miim_if.MTxErr),
    .rx_clk(miim_if.mrx_clk),
    .rx_d(d),
    .rx_dv(miim_if.MRxDV),
    .rx_er(miim_if.MRxErr),
    .col(miim_if.MColl),
    .crs(miim_if.MCrs),
    .half_duplex(1'b0)
);

// Connect interface to testbench virtual interface
string interface_name = (INTERFACE_NAME == "") ? $sformatf("%m") : INTERFACE_NAME;
```

```
initial begin
  uvm_container #(virtual mi_i_if)::set_value_in_global_config(interface_name, miim_if);
end

endmodule
```

It should be noted that if the parameter INTERFACE \_ NAME is not set, then the default value is %m (i.e., the hierarchical path of this module). This is guaranteed to be unique. If this parameter is explicitly set, then it is up to the designer to make sure that the names chosen are unique within the enclosing module.

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# Connect/AbstractConcrete

## Abstract/Concrete Class approach to DUT-TB communication

A handle based approach to DUT-TB communication that does not use virtual interfaces is referred to in the UVM industry as the abstract/concrete class approach. There is also a form of this approach that is in use that is known within Mentor Graphics as Two Kingdoms.

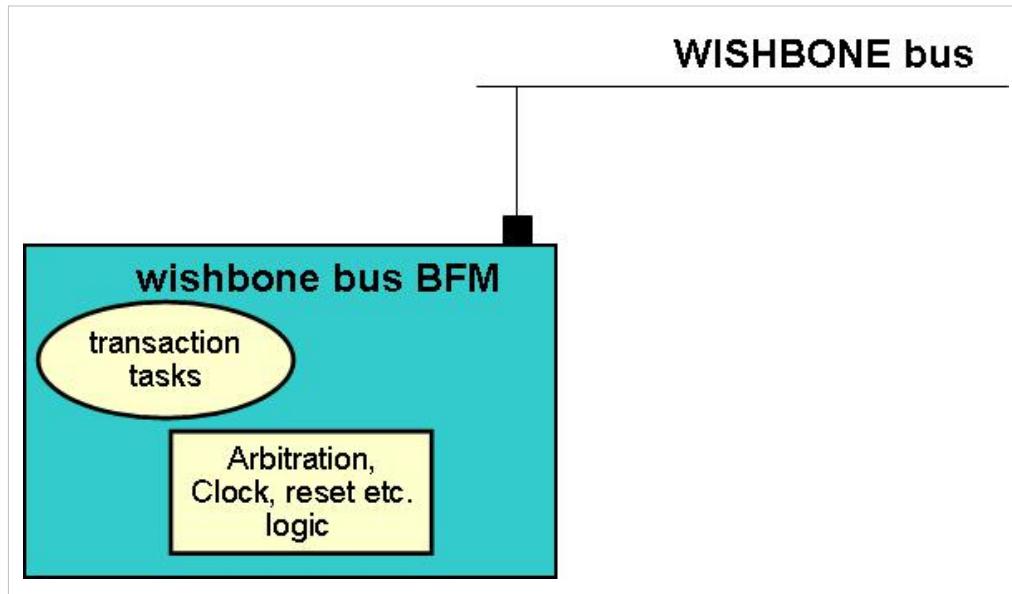
As with using virtual interfaces this approach may be set up for the transactors in the testbench to communicate with the pins of the DUT, with a Bus Functional Model (BFM) which drives transactions on the DUT or the internals of the DUT using the SystemVerilog bind construct. The most typical use is with BFM's.

Virtual interfaces is the recommended approach for DUT-TB communication. The abstract/concrete class approach should only be considered when virtual interfaces can not be used or as a secondary approach in the case of legacy BFM's that can not be modified to be an interface.

The discussion in this article going forward will focus only on use of the abstract/concrete class approach with BFM's.

## Example BFM

Here is a diagram showing a BFM for the WISHBONE bus that will be used in the examples here. The wishbone bus BFM is connected to the WISHBONE bus and has the WISHBONE bus arbitration, clock, reset etc. logic along with tasks which generate WISHBONE bus transactions (read, write etc.).



Here is code from the BFM. The full code may be downloaded with the other example code shown later.

```
module wishbone_bus_syscon_bfm #(int num_masters = 8, int num_slaves = 8,
                                int data_width = 32, int addr_width = 32)
(
  // WISHBONE common signals
  output logic clk,
  output logic rst,
  ...
)
```

```

};

// WISHBONE bus arbitration logic
...
//Slave address decode
...
// BFM tasks
//WRITE 1 or more write cycles
task wb_write_cycle(wb_txn req_txn, bit [2:0] m_id = 1);
...
endtask

//READ 1 or more cycles
task wb_read_cycle(wb_txn req_txn, bit [2:0] m_id = 1, output wb_txn rsp_txn);
...
endtask

// Monitor bus transactions
task monitor(output wb_txn txn);
...
endtask

endmodule

```

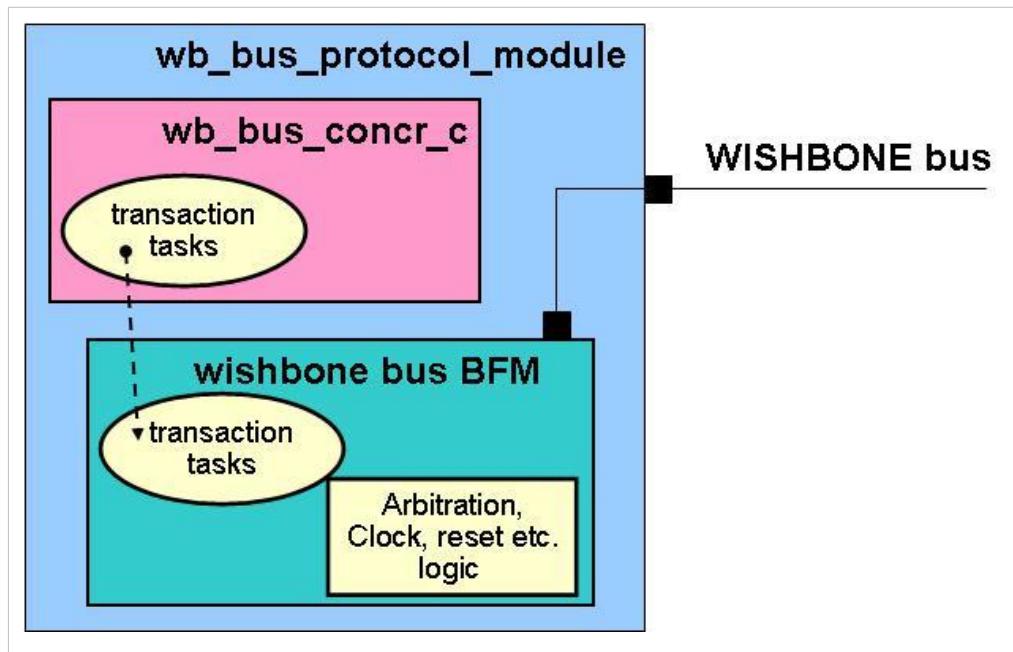
## Abstract/Concrete Classes

First an abstract class (SystemVerilog virtual class) is defined. The abstract class has pure virtual methods and properties which define a public interface for accessing information. The implementations of the methods are not in the abstract class but rather are in a derived class which is referred to as the concrete class. The concrete class is defined inside of a wrapper module which also instantiates the BFM.

## DUT Connection

Since it is defined inside the wrapper module the scope of the concrete class, wb\_bus\_concr\_c in the example, is the wrapper module and hence its methods can access anything defined inside of the wrapper module (wb\_bus\_protocol\_module), including ports, variables, class handles, functions, tasks, module instances etc.

In this diagram an instance of the BFM is created inside the wrapper module. The methods of the concrete class access the BFM methods by hierarchical reference through this instance (bmf\_instance\_name.method\_name).



## Two different use models

After declaration of the abstract class and the concrete class inside the wrapper module there are two use models. There is the recommended use model which follows the abstract/concrete class pattern. Then there is a use model that is called the two kingdoms that is not recommended.

### Abstract/concrete class use model

This approach is recommended as it follows the recommended use model for passing information from the DUT to the testbench which is discussed in detail here in the article on virtual interfaces.

In this use model an actual instance of the concrete class is created inside of the wrapper module where the concrete class is defined. Inside the testbench a base class handle inside of the transactors (drivers, monitors etc.) is made to point to this derived class instance, through use of the config\_db API and the configuration database. More details, diagrams and an example using a BFM are [here](#).

### Two Kingdoms use model

This approach is what is known as the Two Kingdoms. It is *not* recommended as it does not follow the recommended use model for passing information from the DUT to the testbench which is discussed in detail [here](#) in the article on virtual interfaces, instead it uses the factory. Additionally it does not strictly follow the abstract/concrete class pattern with the use of a base class handle in the drivers, monitors etc to point to the concrete class instance but rather the drivers and monitors themselves are the abstract/concrete classes. A factory override is set up of the concrete class for the base class. Inside the testbench a base handle is created and then a factory create is called on the handle resulting in a concrete object being created. More details, diagrams and an example using a BFM are [here](#).

# Connect/AbstractConcreteConfigDB

---

## Abstract Class

The Abstract class is defined as part of the agent in the testbench and is included in the agent package. Below is the code for an example abstract class called wb\_bus\_abs\_c. Note the pure virtual methods which define a public interface to this class. As part of the public interface too is an event to represent the posedge of the clock. Note too that the abstract class inherits from uvm\_component and so inherits the phase methods etc.

```
// Abstract class for abstract/concrete class wishbone bus communication
//-----
virtual class wb_bus_abs_c extends uvm_component;

    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction

    // API methods
    //WRITE 1 or more write cycles
    pure virtual task wb_write_cycle(wb_txn req_txn, bit [2:0] m_id);

    //READ 1 or more cycles
    pure virtual task wb_read_cycle(wb_txn req_txn, bit [2:0] m_id, output wb_txn rsp_txn);

    // wait for an interrupt
    pure virtual task wb_irq(wb_txn req_txn, output wb_txn rsp_txn);

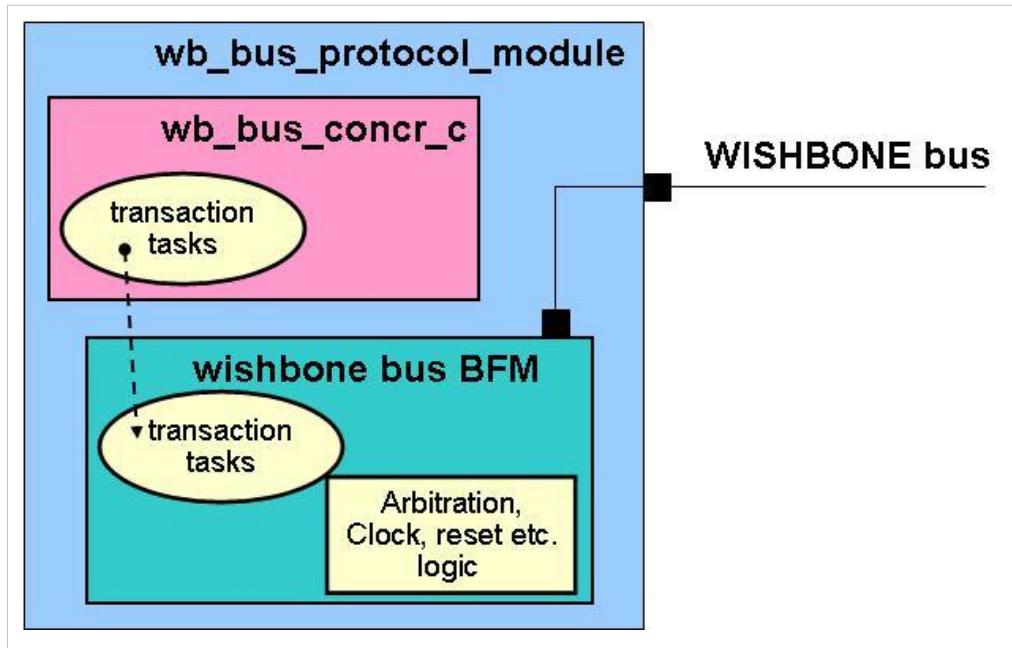
    //Get a wb transaction from the bus
    pure virtual task monitor(output wb_txn txn);

    event pos_edge_clk;

endclass
```

## Concrete Class

The concrete class is derived from the abstract class. It is required to override any pure virtual methods, providing implementations. It may also provide code that writes/reads variables inherited from the abstract class. This class is defined inside of a wrapper module that includes an instance of the BFM. Since it is defined inside the wrapper module the scope of the concrete class, is the wrapper module and hence its methods can access anything defined inside of the wrapper module, including ports, variables, class handles, functions, tasks, and in particular the BFM module instance.



Here is the code for the concrete class `wb_bus_concr_c`. It is defined inside the wrapper module `wb_bus_protocol_module` which is a **protocol module**. Note that this class inherits from the abstract class and provides implementations of the methods. These methods are straight forward in that they are "proxy" methods that simply call the corresponding method inside the BFM. For example the concrete driver class `wb_write_cycle()` task calls the `wb_write_cycle()` task inside the BFM. At the bottom of the `wb_bus_protocol_module` is the instance (`wb_bfm`) of the BFM (`wishbone_bus_syscon_bfm`).

```

module wb_bus_protocol_module #(int WB_ID = 0, int num_masters = 8, int num_slaves = 8,
                           int data_width = 32, int addr_width = 32)
(
  // Port declarations
  // WISHBONE common signals
  output logic clk,
  output logic rst,
  ...
);
  ...
  ...

  // Concrete class declaration
  class wb_bus_concr_c #(int ID = WB_ID) extends wb_bus_abs_c;

    function new(string name = "", uvm_component parent = null);
      super.new(name, parent);
    endfunction

    // API methods
    // simply call corresponding BFM methods
  endclass
endmodule
  
```

```

//WRITE 1 or more write cycles
task wb_write_cycle(wb_txn req_txn, bit [2:0] m_id);
  wb_bfm.wb_write_cycle(req_txn, m_id);
endtask

//READ 1 or more cycles
task wb_read_cycle(wb_txn req_txn, bit [2:0] m_id, output wb_txn rsp_txn);
  wb_bfm.wb_read_cycle(req_txn, m_id, rsp_txn);
endtask

// wait for an interrupt
task wb_irq(wb_txn req_txn, output wb_txn rsp_txn);
  wb_bfm.wb_irq(req_txn, rsp_txn);
endtask

task monitor(output wb_txn txn);
  wb_bfm.monitor(txn);
endtask

task run_phase(uvm_phase phase);
  forever @ (posedge clk)
    -> pos_edge_clk;
endtask : run_phase

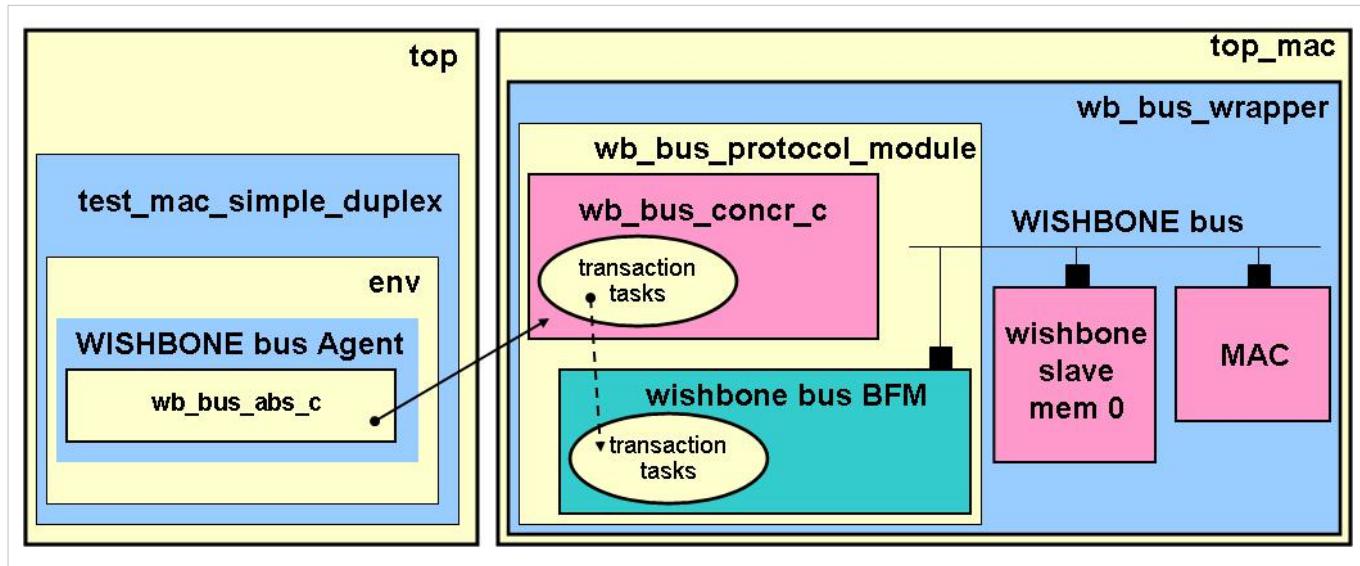
endclass
...
// WISHBONE BFM instance
wishbone_bus_syscon_bfm wb_bfm(
  .clk( clk ),
  .rst( rst ),
  ...
);

```

endmodule

## Connecting the testbench to the DUT

The testbench DUT connection with the abstract and concrete classes is similar to *virtual interface* connections. With a virtual interface connection a virtual interface handle is placed inside the transactor, such as a driver or monitor, and made to point to an instance of an interface connected to the DUT. In the abstract/concrete class approach a handle of the abstract class type is placed inside the transactor. An instance of the concrete class is made inside the wrapper module(wb\_bus\_protocol\_module) and the abstract class handle is made to point to the concrete class instance.



In the diagram above the DUTs and the `wb_bus_protocol_module` are wrapped in a wrapper module the `wb_bus_wrapper`. This is for modularity and convenience in instantiating multiple wishbone buses.

The code below shows the instance of the concrete class inside the wrapper module and a method (a "lazy allocator" ie a method that doesn't allocate the instance until it is needed) used for creating the concrete class instance.

```

module wb_bus_protocol_module #(int WB_ID = 0, int num_masters = 8, int num_slaves = 8,
                           int data_width = 32, int addr_width = 32)
(
  // Port declarations
  // WISHBONE common signals
  output logic clk,
  output logic rst,
  ...
);

  // Concrete class declaration
  class wb_bus_concr_c #(int ID = WB_ID) extends wb_bus_abs_c;
  ...
  endclass

  // instance of concrete class
  wb_bus_concr_c wb_bus_concr_c_inst;

  // lazy allocation of concrete class
  function wb_bus_abs_c get_wb_bus_concr_c_inst();
    if(wb_bus_concr_c_inst == null)
      wb_bus_concr_c_inst = new();
    return (wb_bus_concr_c_inst);
  endfunction

```

```

initial
  //set concrete class object in config space
  uvm_config_db #(wb_bus_abs_c)::set(null, "uvm_test_top",
                                    $sformatf("WB_BUS_CONCR_INST_%0d",WB_ID),
                                    get_wb_bus_concr_c_inst());

  // WISHBONE BFM instance
  wishbone_bus_syscon_bfm wb_bfm(
    .clk( clk ),
    .rst( rst ),
    ...
  );
endmodule

```

The location of the concrete class instance is provided to the transactor in the same manner as in *virtual interface connections* using the `config_db` API to pass a handle that points to the concrete class instance to the test class and then through a configuration object from the test class to the transactor. In the code above inside the initial block a handle to the concrete instance is placed inside the configuration space using the `config_db` API.

In the test class the handle to the concrete driver class instance is fetched from the configuration database and placed inside a configuration object which is made available to the wishbone agent. This approach is recommended as it follows the recommended use model for passing information from the DUT to the testbench which is discussed in detail here in the article on virtual interfaces.

```

class test_mac_simple_duplex extends uvm_test;
  ...
  mac_env env_0;
  wb_config wb_config_0; // config object for WISHBONE BUS 0
  ...

  function void set_wishbone_config_params();
    wb_config_0 = new();
    if (!uvm_config_db #(wb_bus_bfm_driver_base)::get(this, "",
                                                       $sformatf("WB_BUS_BFM_DRIVER_C_INST_%0h", wb_config_0.m_wb_id),
                                                       wb_config_0.m_wb_bfm_driver)) // concrete class object
      `uvm_fatal(report_id, "Can't read WB_BUS_BFM_DRIVER_C_INST");
    ...
    uvm_config_db#(wb_config)::set(this, "env_0*", "wb_config", wb_config_0); // put in config
    ...
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_wishbone_config_params();
    set_mii_config_params();
  endfunction

```

```

...
endfunction : build_phase
...
endclass

```

Inside the driver an abstract class handle is made to point to the concrete class instance by fetching the location from the configuration object provided by the test class.

```

// WISHBONE master driver
class wb_bus_bfm_driver extends uvm_driver #(wb_txn, wb_txn);
`uvm_component_utils(wb_bus_bfm_driver)

uvm_analysis_port #(wb_txn) wb_drv_ap;
bit [2:0] m_id; // Wishbone bus master ID
wb_config m_config;
wb_bus_abs_c m_wb_bus_abs_c;

function new(string name, uvm_component parent);
super.new(name,parent);
endfunction

function void build_phase(uvm_phase phase);
super.build_phase(phase);
if (!uvm_config_db #(wb_config)::get(this, "", "wb_config", m_config)) // get config object
`uvm_fatal("Config Fatal", "Can't get the wb_config")
m_id = m_config.m_wb_master_id;
wb_drv_ap = new("wb_drv_ap", this);
// Assign abstract class handle to concrete object
m_wb_bus_abs_c = m_config.m_wb_bus_abs_c;
endfunction : build_phase

task run_phase(uvm_phase phase);
wb_txn req_txn;
forever begin
seq_item_port.get(req_txn); // get transaction
@ ( m_wb_bus_abs_c.pos_edge_clk) #1; // sync to clock edge + 1 time step
case(req_txn.txn_type) //what type of transaction?
NONE: `uvm_info($sformatf("WB_M_DRVR_%0d",m_id),
$ssformatf("wb_txn %0d the wb_txn_type was type NONE",
req_txn.get_transaction_id()), UVM_LOW)
WRITE: wb_write_cycle(req_txn);
READ: wb_read_cycle(req_txn);
RMW: wb_rmw_cycle(req_txn);
WAIT_IRQ: fork wb_irq(req_txn); join_none
default: `uvm_error($sformatf("WB_M_DRVR_%0d",m_id),

```

```

        $sformatf("wb_txn %0d the wb_txn_type was type illegal",
        req_txn.get_transaction_id() ) )

      endcase
    end
  endtask : run_phase

  //READ 1 or more cycles
  virtual task wb_read_cycle(wb_txn req_txn);
    wb_txn rsp_txn;
    m_wb_bus_abs_c.wb_read_cycle(req_txn, m_id, rsp_txn);
    seq_item_port.put(rsp_txn); // send rsp object back to sequence
    wb_drv_ap.write(rsp_txn); //broadcast read transaction with results
  endtask

  //WRITE 1 or more write cycles
  virtual task wb_write_cycle(wb_txn req_txn);
    wb_txn orig_req_txn;
    $cast(orig_req_txn, req_txn.clone()); //save off copy of original req transaction
    m_wb_bus_abs_c.wb_write_cycle(req_txn, m_id);
    wb_drv_ap.write(orig_req_txn); //broadcast original transaction
  endtask

  //RMW ( read-modify-write)
  virtual task wb_rmw_cycle(ref wb_txn req_txn);
    `uvm_info($sformatf("WB_M_DRVR_%0d",m_id),
              "Wishbone RMW instruction not implemented yet",UVM_LOW )
  endtask

  // wait for an interrupt
  virtual task wb_irq(wb_txn req_txn);
    wb_txn rsp_txn;
    m_wb_bus_abs_c.wb_irq(req_txn, rsp_txn);
    seq_item_port.put(rsp_txn); // send rsp object back to sequence
  endtask
endclass

```

When the driver receives a WISHBONE write transaction for example in the run task it calls its wb\_write\_cycle() task which uses the abstract class handle (m\_wb\_bus\_abs\_c) to call the wb\_write\_cycle() method in the concrete class which in turn calls the wb\_write\_cycle() method in the BFM.

## Multiple instance considerations

When there are multiple instances of the DUT & BFM then each concrete class instance needs to be "uniquified" so the correct pairing can be made. A way to do this is to parameterize the wrapper class and the concrete class. In this example an integer value which is the WISHBONE bus id (WB\_ID) is supplied to make a unique string with. This is done to make a correlation between bus id's and the concrete classes. Another approach to making the concrete class instance unique is described in the article on *protocol modules* (see at the bottom). The full source code in this article actually has two wishbone wrapper class instances inside of top\_mac instead of the one shown in the diagrams and so there are two WISHBONE buses, MACs etc. On the testbench side it has a wishbone environment for each of the WISHBONE buses instead of just one as shown in the diagrams. Separate configuration objects are generated for each of the wishbone environments. The details for the multiple instances are not shown in this article but can be viewed in the the source code.

( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

# Configuring a Test Environment

## Configuration

Learn about passing configuration into a UVM test environment

### Configuration Chapter contents:

[Configuration \(this page\)](#) - how to use configuration values and config objects

[Config/ConfiguringSequences](#) - how to configure sequences

[Resources/config db](#) - how to use `uvm_config_db` to store and load configuration objects

[Config/Params Package](#) - good practice is to encapsulate all config parameters in a package for easy access from outside and inside simulation

[ParametersAndReuse](#) - an advanced article, which discusses how to pass parameters down through the testbench

[Config/Container](#) - the OVM container solution is not recommended by Mentor - use the `uvm_config_db` API instead

[Config/SetGetConfig](#) - the OVM set/get config solution is not recommended by Mentor - use the `uvm_config_db` API

## Topic Overview

### Introduction

One of the key tenets of designing reusable testbenches is to make testbenches as configurable as possible. Doing this means that the testbench and its constituent parts can easily be reused and quickly modified.

In a testbench, there are any number of values that you might normally write as literals - values such as for-loop limits, string names, randomization weights and other constraint expression values, coverage bin values. These values can be represented by SystemVerilog variables, which can be set (and changed) at runtime, or SystemVerilog parameters, which must be set at compile time. Because of the flexibility they offer, variables organised into configuration objects and accessed using the `uvm_config_db` API should always be used where possible.

However, bus widths have to be fixed at compile time, so cannot be implemented as configuration objects. There are a number of articles on handling parameters in the UVM :

- [Parameterized Tests](#) shows how to use parameterized tests with the UVM factory
- [The test parameter package article](#) shows how to centralize the parameters shared between DUT and testbench
- [The Parameters and reuse article](#) shows how to pass large numbers of parameters down through the `uvm_component` hierarchy.

## Configuration Objects

Configuration objects are an efficient, reusable mechanism for organizing configuration variables. In a typical testbench, there will in general be several configuration objects, each tied to a component. They are created as a subclass of `uvm_object` and group together all related configuration parameters for a given branch of the test structural hierarchy. There can also be an additional, single configuration object that holds global configuration parameters.

The UVM configuration database takes care of the scope and storage of the object. Below is the code for a typical configuration object for an agent. It has a virtual interface, which is used to point to the interface that the agent is connected to, and a number of variables used to describe and control that interface.

```
// configuration class
class wb_config extends uvm_object;
  `uvm_object_utils( wb_config );

// Configuration Parameters
  virtual wishbone_bus_syscon_if v_wb_bus_if; // virtual wb_bus_if

  int m_wb_id;                                // Wishbone bus ID
  int m_wb_master_id;                          // Wishbone bus master id for wishbone agent
  int m_mac_id;                               // id of MAC WB master
  int unsigned m_mac_wb_base_addr;             // Wishbone base address of MAC
  bit [47:0] m_mac_eth_addr;                  // Ethernet address of MAC
  bit [47:0] m_tb_eth_addr;                   // Ethernet address of testbench for sends/receives
  int m_mem_slave_size;                      // Size of slave memory in bytes
  int unsigned m_s_mem_wb_base_addr;           // base address of wb memory for MAC frame buffers
  int m_mem_slave_wb_id;                      // Wishbone ID of slave memory
  int m_wb_verbosity;                         // verbosity level for wishbone messages

  function new( string name = "" );
    super.new( name );
  endfunction

endclass
```

## Using a Configuration Object

Any component that requires configuration should perform the following steps:

- get its own configuration
- create its own internal structure and behavior based on its configuration
- configure its children

The test component, as the top-level component, gets its configuration values from either a test parameter package or from the UVM configuration database (e.g. for a virtual interface handle). It then sets test-specific configuration parameters for components in the environment.

```

class test_mac_simple_duplex extends uvm_test;
  ...

  wb_config wb_config_0; // config object for WISHBONE BUS
  ...

  function void set_wishbone_config_params();
    //set configuration info
    // NOTE The MAC is WISHBONE slave 0, mem_slave_0 is WISHBONE slave 1
    // MAC is WISHBONE master 0, wb_master is WISHBONE master 1
    wb_config_0 = new();

    if( !uvm_config_db #(virtual wishbone_bus_syscon_if)::get( this , "" , "WB_BUS_IF" , wb_config_0.v_wb_bus_if ) ) begin
      `uvm_error(...)
    end

    wb_config_0.m_wb_id = 0; // WISHBONE 0
    wb_config_0.m_mac_id = 0; // the ID of the MAC master
    wb_config_0.m_mac_eth_addr = 48'h000BC0D0EF00;
    wb_config_0.m_mac_wb_base_addr = 32'h00100000;
    wb_config_0.m_wb_master_id = 1; // the ID of the wb master
    wb_config_0.m_tb_eth_addr = 48'h000203040506;
    wb_config_0.m_s_mem_wb_base_addr = 32'h00000000;
    wb_config_0.m_mem_slave_size = 32'h00100000; // 1 Mbyte
    wb_config_0.m_mem_slave_wb_id = 0; // the ID of slave mem
    wb_config_0.m_wb_verbosity = 350;

    uvm_config_db #( wb_config )::set( this , "*" , "wb_config" , wb_config_0 );
  endfunction
  ...

  function void build_phase( uvm_phase );
    super.build_phase( phase );

    set_wishbone_config_params();
    ...
  endfunction
  ...

endclass

```

The components that use the configuration object get it by using `uvm_config_db::get`. In this example, the drivers get the virtual interface handle, ID, and verbosity from the object.

```

class wb_m_bus_driver extends uvm_driver #(wb_txn, wb_txn);
  ...

  virtual wishbone_bus_syscon_if m_v_wb_bus_if;
  bit [2:0] m_id; // Wishbone bus master ID
  wb_config m_config;
  ...

  function void build_phase( uvm_phase phase );
    super.build_phase( phase );

    if( !uvm_config_db #( wb_config )::get( this , "" , "wb_config" , m_config ) ) begin
      `uvm_error(...)

    end
    m_id = m_config.m_wb_master_id;
    ...
  endfunction

  function void connect_phase( uvm_phase phase );
    super.connect_phase( phase );
    m_v_wb_bus_if = m_config.v_wb_bus_if; // set local virtual if property
  endfunction

  function void end_of_elaboration();
    set_report_verbosity_level_hier(m_config.m_wb_wb_verbosity);
  endfunction

  ...
endclass

```

## Configuring sequences

There is a separate article on Configuring Sequences here.

## Configuring DUT connections

Setting up DUT-to-Testbench connections is one kind of configuration activity that is always necessary. An SV module ( usually the top level module but sometimes a ProtocolModules ) has to add a virtual interface into the configuration space. On the testbench side the test component gets the virtual interface handle from the UVM config database and applies it to appropriate configuration objects:

```

class test_mac_simple_duplex extends uvm_test;
  ...

  function void set_wishbone_config_params();
    wb_config_0 = new();

```

```
// Get the virtual interface handle that was set in the top module or protocol module
if( !uvm_config_db #( virtual wishbone_bus_syscon_if )::get( this , "" , "WB_BUS_IF" , wb_config_0.v_wb_bus_if ) ) begin
  `uvm_error(...)

end

...
uvm_config_db #( wb_config )::set( this , "*", "wb_config", wb_config_0, 0); // put in config
endfunction
...
endclass
```

## Example source code

( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

# Resources/config db

The `uvm_config_db` class is the recommended way to access the resource database. A resource is any piece of information that is shared between more than one component or object. We use `uvm_config_db::set` to put something into the database and `uvm_config_db::get` to retrieve information from the database. The `uvm_config_db` class is parameterized, so the database behaves as if it is partitioned into many type-specific "mini databases." There are no limitations on the type - it could be a class, a `uvm_object`, a built in type such as a bit, byte, or a virtual interface.

There are two typical uses for `uvm_config_db`. The first is to pass virtual interfaces from the DUT to the test, and the second is to pass configuration classes down through the testbench.

## The set method

The full signature of the set method is `void uvm_config_db #( type T = int )::set( uvm_component ctxt , string inst_name , string field_name , T value );`

- `T` is the type of the element being configured - usually a virtual interface or a configuration object.
- `ctxt` and `inst_name` together form a scope that is used to locate the resource within the database. The scope is formed by appending the instance name to the full hierarchical name of the context, i.e.  
`{ctxt.get_full_name(),".",inst_name}`.
- `Field_name` supplies the name of the resource
- `value` is the thing which is actually going to be put into the database.

An example of putting virtual interfaces into the configuration database is shown below:

```
interface ahb_if data_port_if( clk , reset );
interface ahb_if control_port_if( clk , reset );
...
uvm_config_db #( virtual ahb_if )::set( null , "uvm_test_top" , "data_port" , data_port_if );
uvm_config_db #( virtual ahb_if )::set( null , "uvm_test_top" , "control_port" , control_port_if );
```

This code puts two AHB interfaces into the configuration database at the hierarchical location "uvm\_test\_top", which is the default location for the top level test component created by `run_test()`. The two different interfaces are put into the database using two distinct names, "data\_port" and "control\_port".

Notes:

- We use "`uvm_test_top`" because it is more precise and more efficient than "\*". This will always work unless your top level test does something other than `super.new( name , parent )` in the constructor of your test component. If you do something different in your test component, you will have to adjust the instance name accordingly.
- We use "`null`" in the first argument because this code is in a top level module not a component.

An example of configuring agents inside an env is shown below:

```
class env extends uvm_env;
//
ahb_agent_config ahb_agent_config_h;
//
function void build_phase( uvm_phase phase );
...

```

```

m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent" , this );
...
uvm_config_db #( ahb_agent_config )::set( this , "m_ahb_agent*" , "ahb_agent_config" , ahb_agent_config_h );
...
endfunction
endclass

```

This code sets the configuration for the ahb agent and all its children. Two things to note about this code are:

- We use "**this**" as our first argument, so we can be sure that we have only configured this env's agent and not any other ahb agent in the component hierarchy
- We use "**m\_ahb\_agent\***" to ensure that we config both the agent and its children. Without the '\*' we would only configure the agent itself and not the driver, sequencer and monitor inside it.

## The get method

The full signature of the get method is **bit uvm\_config\_db #( type T = int )::get( uvm\_component ctxt , string inst\_name , string field\_name , ref T value );**

- **T** is the type of the element being configured - usually a virtual interface or a configuration object.
- **ctxt** and **inst\_name** together form a scope that is used to locate the resource within the database. The scope is formed by appending the instance name to the full hierarchical name of the context, i.e.  
`{ctxt.get_full_name(),".",inst_name}.`
- **Field\_name** supplies the name of the resource
- **value** is the thing which is going to be retrieved from the database. If the call to get is successful, this value will be overwritten.
- The method **returns** 1 if it is successful and 0 if there is no such resource of this type at this location.

An example of getting virtual interfaces from the configuration database is shown below:

```

class test extends uvm_test;
...
function void build_phase( uvm_phase phase );
...
if( !uvm_config_db #( virtual ahb_if )::get( this , "" , "data_port" , m_cfg.m_data_port_config.m_ahb_if ) ) begin
`uvm_error("Config Error" , "uvm_config_db #( virtual ahb_if )::get cannot find resource data_port" )
end
...
endfunction
...
endclass

```

The code attempts to get the virtual interface for the AHB dataport and put it into the correct agent's configuration class. If this attempt does not succeed, we provide a meaningful error message.

An example of getting a configuration class in a transactor is shown below:

```

class ahb_monitor extends uvm_monitor;
ahb_agent_config m_cfg;

```

```
function void build_phase( uvm_phase phase );
  ...
  if( !uvm_config_db #( ahb_agent_config )::get( this , "" , "ahb_agent_config" , m_cfg ) ) begin
    `uvm_error("Config Error" , "uvm_config_db #( ahb_agent_config )::get cannot find resource ahb_agent_config" )
  end
  ...
endfunction
endclass
```

Notes:

- We always use **this** as the context
- We usually use "" as the instance name
- We always check to see if the call to get has succeeded. If it doesn't, we produce a reasonable error message.

## Precedence Rules

There are two sets of precedence rules that apply to `uvm_config_db`. The first is that during the build phase, a set in a context higher up the component hierarchy takes precedence over a set which occurs lower down the hierarchy. The second is that when the context is the same or we have finished with the build phase, then the last set takes precedence over an earlier one. For more detail on these rules, please consult the reference manual available from Accellera or look directly at the comments in the UVM code.

# Config/Params Package

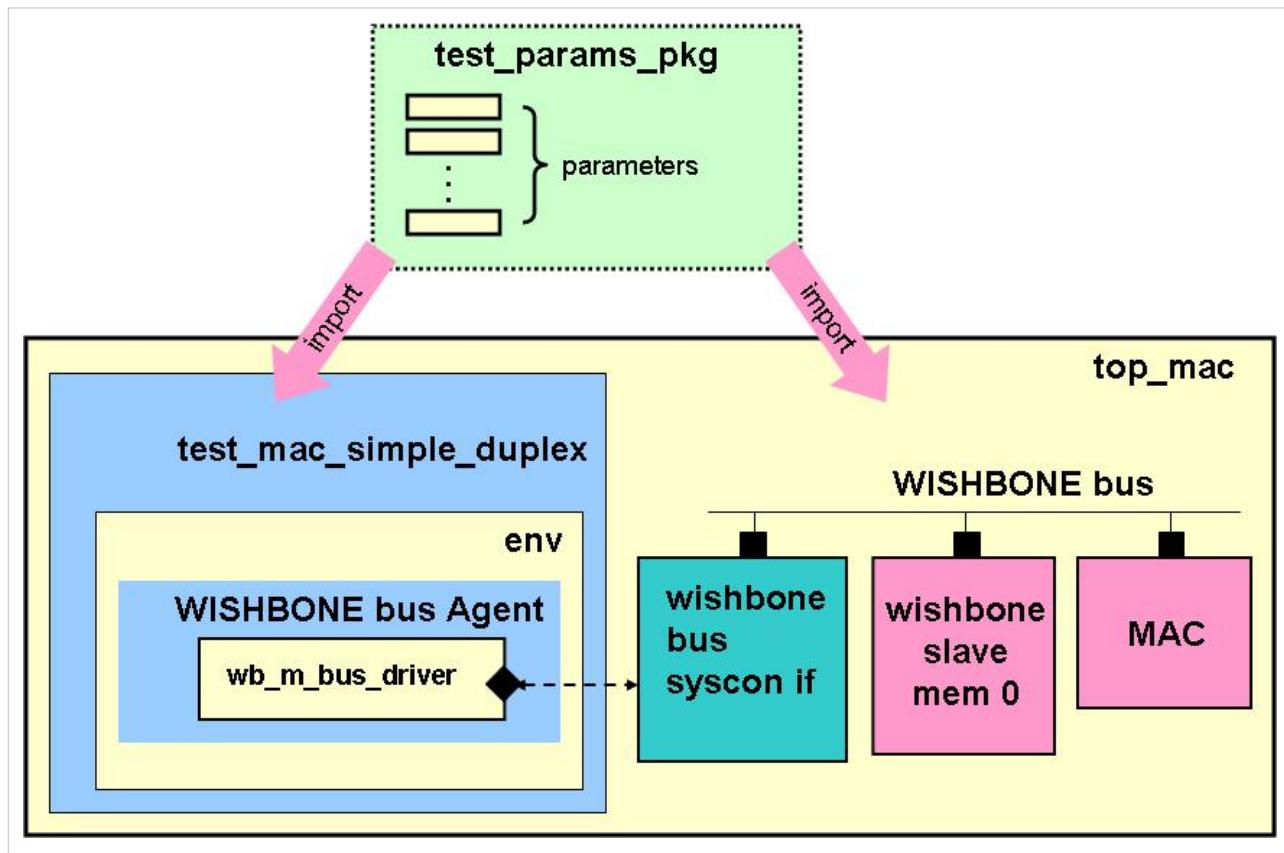
When a DUT and/or interface is parameterized the parameter values are almost always used in the testbench as well. These parameters should not be specified with direct literal values in your instantiations. Instead define named parameter values in a package and use the named values in both the DUT side as well as the testbench side of the design.

This helps avoid mistakes where a change is made to one side but not to the other. Or, if a test configuration parameter is some function of a DUT parameter, there is a chance that a miscalculation may occur when making a change.

Note that this package is not a place for all test parameters. If you have test-specific parameters that are not used by the DUT, those values should be set directly in the test. The DUT parameters package should be used only for parameters that are shared between the DUT and the test.

## Example use of a Parameter Package

In the WISHBONE bus based example below there are two WISHBONE bus devices. A slave memory and an Ethernet Media Access Controller (MAC). Parameters are placed in a package (test\_params\_pkg) and then used in instantiating the WISHBONE devices in the top level module and inside the test class of the testbench.



The test parameter package is shown below:

```
package test_params_pkg;
import uvm_pkg::*;

// WISHBONE general slave parameters
```

```

parameter slave_addr_space_sz = 32'h00100000;

// WISHBONE slave memory parameters
parameter mem_slave_size = 18; // 2**mem_slave_size = size in words(32 bits) of wb slave memory
parameter mem_slave_wb_id = 0; // WISHBONE bus slave id of wb slave memory

// MAC WISHBONE parameters
parameter mac_m_wb_id = 0; // WISHBONE bus master id of MAC
parameter mac_slave_wb_id = 1; // WISHBONE bus slave id of MAC

endpackage

```

The parameter values (mem\_slave\_size, mem\_slave\_wb\_id) usage in the top module to instantiate the WISHBONE bus slave memory module is shown below. Note the import of the test\_params\_pkg in the top\_mac module:

```

module top_mac;
...
import test_params_pkg::*;

// WISHBONE interface instance
// Supports up to 8 masters and up to 8 slaves
wb_bus_if wb_bus_if();

//-----
// WISHBONE 0, slave 0: 000000 - 0fffff
// this is 1 Mbytes of memory
wb_slave_mem #(mem_slave_size) wb_s_0 (
    // inputs
    .clk ( wb_bus_if.clk ),
    .rst ( wb_bus_if.rst ),
    .adr ( wb_bus_if.s_addr ),
    .din ( wb_bus_if.s_wdata ),
    .cyc ( wb_bus_if.s_cyc ),
    .stb ( wb_bus_if.s_stb[mem_slave_wb_id] ),
    .sel ( wb_bus_if.s_sel[3:0] ),
    .we ( wb_bus_if.s_we ),
    // outputs
    .dout( wb_bus_if.s_rdata[mem_slave_wb_id] ),
    .ack ( wb_bus_if.s_ack[mem_slave_wb_id] ),
    .err ( wb_bus_if.s_err[mem_slave_wb_id] ),
    .rty ( wb_bus_if.s_rty[mem_slave_wb_id] )
);

...
endmodule

```

Parameter usage inside the test class of the testbench to set the configuration object values for the WISHBONE bus slave memory is shown below. Note that instead of using the numeric literal of 32'h00100000 for the address value, the code uses an expression involving a DUT parameter (mem\_slave\_size).

```
package tests_pkg;
...
import test_params_pkg::*;

`include "test_mac_simple_duplex.svh"

endpackage

//-----
class test_mac_simple_duplex extends uvm_test;
...
wb_config wb_config_0; // config object for WISHBONE BUS
...

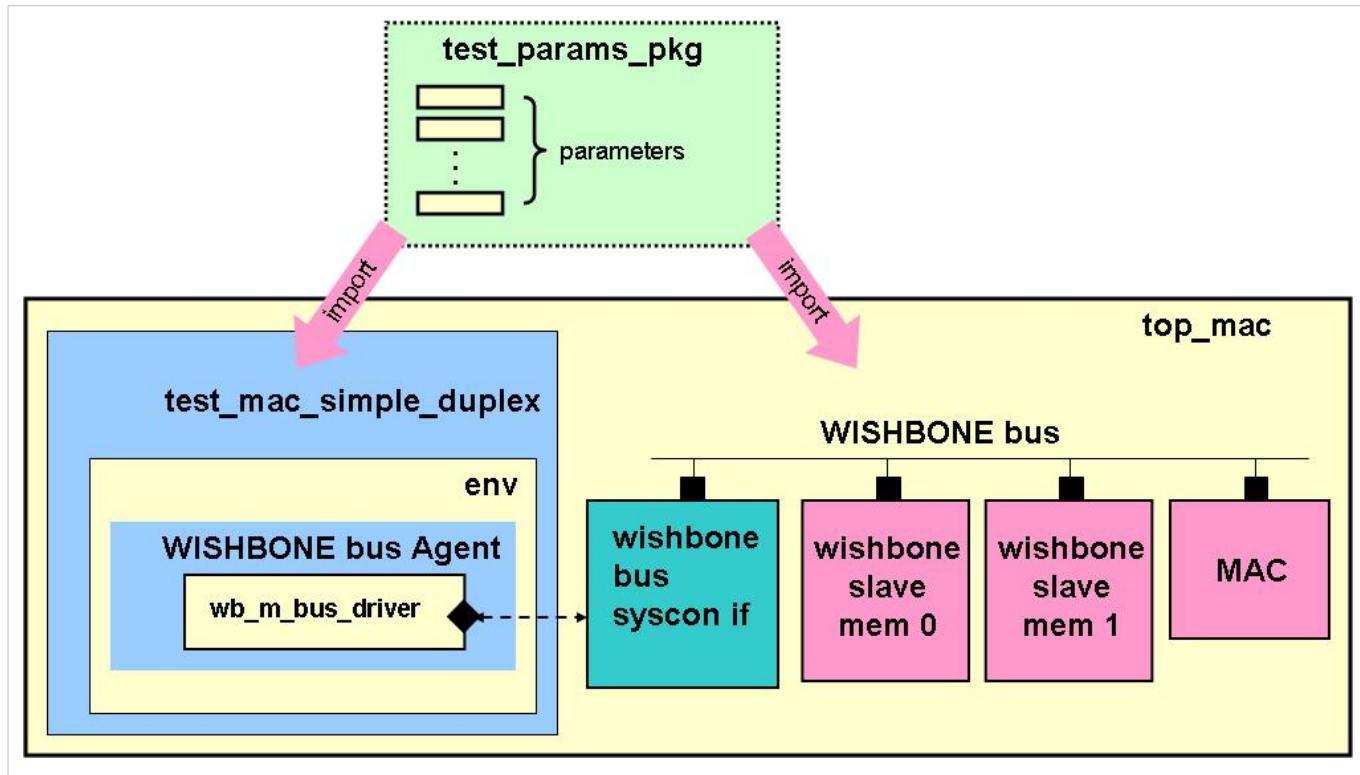
function void set_wishbone_config_params();
  //set configuration info
  wb_config_0 = new();

  wb_config_0.m_s_mem_wb_base_addr = mem_slave_wb_id * slave_addr_space_sz; // base address of slave mem
  wb_config_0.m_mem_slave_size = 2** (mem_slave_size+2); // default is 1 Mbyte
  wb_config_0.m_mem_slave_wb_id = mem_slave_wb_id; // WISHBONE bus slave id of slave mem
  ...
endfunction
...

endclass
```

## Multiple Instances

When you have multiple instances of parameter sets you can either create a naming convention for your parameters or you can use a parameterized class-based approach to organize your parameter sets on a per-instance basis.



Create a parameterized class which specifies the parameters and their default values. Then for each instance set the parameter values by creating a specialization of the parameterized class using a typedef.

```

package test_params_pkg;
import uvm_pkg::*;

// WISHBONE general slave parameters
parameter slave_addr_space_sz = 32'h00100000;

// WISHBONE slave memory parameters
class WISHBONE_SLAVE #(int mem_slave_size = 18, int mem_slave_wb_id = 0);
endclass

// Specializations for each slave memory instance
typedef WISHBONE_SLAVE #(18, 0) WISHBONE_SLAVE_0;
typedef WISHBONE_SLAVE #(18, 1) WISHBONE_SLAVE_1;

// MAC WISHBONE parameters
parameter mac_m_wb_id = 0;          // WISHBONE bus master id of MAC
parameter mac_slave_wb_id = 2;      // WISHBONE bus slave id of MAC

```

```
endpackage
```

To access or use the parameters, such as `mem_slave_size` or `mem_slave_wb_id`, specified in the specializations `WISHBONE_SLAVE_0` or `WISHBONE_SLAVE_1` in the above code, use the following syntax `name_of_specialization::parameter_name` as illustrated below.

```
module top_mac;
...
import test_params_pkg::*;

// WISHBONE interface instance
// Supports up to 8 masters and up to 8 slaves
wishbone_bus_syscon_if wb_bus_if();

//-----
// WISHBONE 0, slave 0: 000000 - 0fffff
// this is 1 Mbytes of memory
wb_slave_mem #(WISHBONE_SLAVE_0::mem_slave_size) wb_s_0 (
    // inputs
    .clk ( wb_bus_if.clk ),
    .rst ( wb_bus_if.rst ),
    .adr ( wb_bus_if.s_addr ),
    .din ( wb_bus_if.s_wdata ),
    .cyc ( wb_bus_if.s_cyc ),
    .stb ( wb_bus_if.s_stb [WISHBONE_SLAVE_0::mem_slave_wb_id] ),
    .sel ( wb_bus_if.s_sel[3:0] ),
    .we ( wb_bus_if.s_we ),
    // outputs
    .dout( wb_bus_if.s_rdata[WISHBONE_SLAVE_0::mem_slave_wb_id] ),
    .ack ( wb_bus_if.s_ack [WISHBONE_SLAVE_0::mem_slave_wb_id] ),
    .err ( wb_bus_if.s_err [WISHBONE_SLAVE_0::mem_slave_wb_id] ),
    .rty ( wb_bus_if.s_rty [WISHBONE_SLAVE_0::mem_slave_wb_id] )
);

...
endmodule
```

There is further discussion of the relationship between parameters and reuse here.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Config/ConfiguringSequences

## A Configurable Sequence

The most generic way to configure sequences is to use the full hierarchical name by default, but allow any other name to be used by the creator of the sequence.

```
class my_bus_seq extends uvm_sequence #( my_bus_sequence_item );
  string scope_name = "";

  task body();
    my_bus_config m_config;

    if( scope_name == "" ) begin
      scope_name = get_full_name(); // this is { sequencer.get_full_name() , get_name() }
    end

    if( !uvm_config_db #( my_bus_config )::get( null , scope_name , "my_bus_config" , m_config ) ) begin
      `uvm_error(...)
    end
  endtask
endclass
```

Suppose that we have a sequence called "initialization\_sequence" running on the sequencer on "uvm\_test\_top.env.sub\_env.agent1.sequencer". This scope\_name in the code above by default is "uvm\_test\_top.env.sub\_env.agent1.sequencer.initialization\_sequence".

The most usual use case for sequence configuration is that we want to pick up the agent's configuration class which has been set on the agent and all its children.

```
class sub_env extends uvm_env;
  ...
  function void build_phase( uvm_phase phase );
    ...
    my_bus_config agent1_config;
    ...
    uvm_config_db #( my_bus_config )::set( this , "agent1*" , "my_bus_config" , agent1_config );
    ...
  endfunction

  task main_phase( uvm_phase phase );
    my_bus_sequence seq = my_bus_sequence::type_id::create("my_bus_sequence");

    seq.start( agent1.sequencer );
  endtask
  ...

```

```
endclass
```

Since we have used the sub\_env as the context, the set and the default get in the configurable sequence will match and as a result the sequence will have access to the agent's configuration object.

## Per Sequence Configuration

We can use the default mode of the configurable sequence above to configure distinct sequences differently, although we can only do this if the names of the sequences are different.

For example, the environment class might look something like this:

```
class sub_env extends uvm_env;
  ...
  function void build_phase( uvm_phase phase );
    ...
    my_bus_config agent1_config , agent1_error_config;
    ...
    agent1_config.enable_error_injection = 0;
    agent1_error_config.enable_error_injection =10;

    // most sequences do not enable error injection
    uvm_config_db #( my_bus_config )::set( this , "agent1*" , "my_bus_config" , agent1_config );

    // sequences with "error" in their name will enable error injection
    uvm_config_db #( my_bus_config )::set( this , "agent1.sequencer.error*" , "my_bus_config" , agent1_error_config );
    ...
  endfunction

  task main_phase( uvm_phase phase );
    my_bus_sequence normal_seq = my_bus_sequence::type_id::create("normal_seq");
    my_bus_sequence error_seq = my_bus_sequence::type_id::create("error_seq");

    normal_seq.start( agent1.sequencer );
    error_seq.start( agent1.sequencer );
  endtask
  ...
endclass
```

Since the configurable sequence uses the sequence name to do a get, the normal sequence will pick up the configuration which does not enable error injection while the error sequence will pick up the error configuration.

## Ignoring the Component Hierarchy Altogether

It is quite possible to completely ignore the component hierarchy when configuring sequences. This has the advantage that we can in effect define behavioural scopes which are only intended for configuring sequences, and we can keep these behavioural scopes completely separate from the component hierarchy. The configurable sequence described above will work in this context as well as those above which are based on the component hierarchy.

So for example in a virtual sequence we might do something like:

```
class my_virtual_sequence extends uvm_sequence #( uvm_sequence_item_base );  
  ...  
  task body();  
    my_bus_sequence normal_seq = my_bus_sequence::type_id::create("normal_seq");  
    my_bus_sequence error_seq = my_bus_sequence::type_id::create("error_seq");  
  
    normal_seq.scope_name = "sequences::my_bus_config.no_error_injection";  
    error_seq.scope_name = "sequences::my_bus_config.enable_error_injection";  
  
    normal_seq.start( agent1.sequencer );  
    error_seq.start( agent1.sequencer );  
  endtask  
  ...  
endclass
```

The downside to this freedom is that every sequence and component in this testbench has to agree on the naming scheme, or at least be flexible enough to deal with this kind of arbitrary naming scheme. Since there is no longer any guarantee of uniqueness for these scope names, there may be some reuse problems when moving from block to integration level.

# ResourceAccessForSequences

Sequences often need access to testbench resources such as register models or configuration objects. The way to do this is to use the `uvm_config_db` to access a resource, and to put the access functionality into the `body()` method of sequence base class which is used by other sequences extended from the base class.

There are several ways in which the `uvm_config_db` can be used to access the resource:

- Access the resource using the `m_sequencer` handle

```
// Resource access using m_sequencer:
spi_env_config m_cfg;

task body();
  if(!uvm_config_db #(spi_env_config)::get(m_sequencer, "", "spi_env_config", m_cfg)) begin
    `uvm_error("BODY", "spi_env_config config_db lookup failed")
  end

endtask: body
```

- Access the resource using the sequence's `get_full_name()` call

```
// Resource access using get_full_name():
spi_env_config m_cfg;

task body();
  if(!uvm_config_db #(spi_env_config)::get(null, get_full_name(), "spi_env_config", m_cfg)) begin
    `uvm_error("BODY", "spi_env_config config_db lookup failed")
  end

endtask: body
```

- Access the resource using a scope string when the resource is not tied to the component hierarchy

```
// Resource access using pre-assigned lookup:
spi_env_config m_cfg;

task body();
  if(!uvm_config_db #(spi_env_config)::get(null, "SPI_ENV::", "spi_env_config", m_cfg)) begin
    `uvm_error("BODY", "spi_env_config config_db lookup failed")
  end

endtask: body
```

The first two methods are mostly equivalent, since they are creating a scope string based either on the `m_sequencer`'s place in the testbench hierarchy or the sequence's psuedo place in the testbench hierarchy. The difference between the first method and the second method is when `m_sequencer` is passed in as the argument to the `get()` function, the `get()` function will call `m_sequencer.get_full_name()`. This results in a path to the sequencer in the testbench heirarchy. An

example of this might be "uvm\_test\_top.env.my\_agent.sequencer". The second method is calling get\_full\_name() on the sequence. If the sequence has been started on a sequencer (m\_sequencer handle is not null), then the sequence's get\_full\_name() call returns the path to the sequencer with the sequence's name appended to the end. An example of this might be "uvm\_test\_top.env.my\_agent.sequencer.my\_seq". This could be useful to target specific configuration information at a specific sequence running on a specific sequencer. The third method is based on users agreeing to a naming convention for different configuration domains, this may work within a particular project or workgroup but may cause issues with reuse due to name clashes.

A full example of an implementation of a base sequence is shown below. Any inheriting sequences would then call the base sequences body() method to ensure that resource handles were set up before starting their stimulus process.

```
//  
// Sequence that needs to access a test bench resource via  
// the configuration space  
  
//  
// Note that this is a base class any class extending it would  
// call super.body() at the start of its body task to get set up  
  
//  
class register_base_seq extends uvm_sequence #(bus_seq_item);  
  
`uvm_object_utils(register_base_seq)  
  
// Handle for the actual sequencer to be used:  
bus_sequencer BUS;  
  
// Handle for the environment configuration object:  
bus_env_config env_cfg;  
  
// Handle for the register model  
dut_reg_model RM;  
  
function new(string name = "register_base_seq");  
  super.new(name);  
endfunction  
  
task body;  
  // Get the configuration object for the env - using get_full_name()  
  if(!uvm_config_db #(bus_env_config)::get(null, get_full_name(), "bus_config", env_cfg)) begin  
    `uvm_error("BODY", "Failed to find bus_env_config in the config_db")  
  end  
  // Assign a pointer to the register model which is inside the env config object:  
  RM = env_cfg.register_model;  
endtask: body  
  
endclass: register_base_seq
```

```
//  
// An inheriting sequence:  
//  
class initialisation_seq extends register_base_seq;  
  
'uvm_object_utils(initialisation_seq)  
  
task body;  
    super.body(); // assign the resource handles  
    //  
    // Sequence body code  
    //  
endtask: body  
  
endclass: initialisation_seq
```

Another example of using this technique is to access a virtual interface within a configuration object to wait for hardware events.

# MacroCostBenefit

Macros should be used to ease repetitive typing of small bits of code, to hide implementation differences or limitations among the vendors' simulators, or to ensure correct operation of critical features. Many of the macros in OVM meet these criterion, while others clearly do not. While the benefits of macros may be obvious and immediate, the costs associated with using certain macros are hidden and may not be realized until much later.

This topic is explored in the following paper from DVCon11:

OVM-UVM Macros-Costs vs Benefits DVCon11 Appnote (PDF) <sup>[1]</sup>

This paper will:

- Examine the hidden costs incurred by some macros, including code bloat, low performance, and debug difficulty.
- Identify which macros provide a good cost-benefit trade-off, and which do not.
- Teach you how to replace high-cost macros with simple SystemVerilog code.

Summary Recommendations

Macro Type	Description
<code>`ovm_*_utils</code>	Always use. These register the object or component with the OVM factory. While not a lot of code, registration can be hard to debug if not done correctly.
<code>`ovm_info</code> <code>`ovm_warning</code> <code>`ovm_error</code> <code>`ovm_fatal</code>	Always use. These can significantly improve performance over their function counterparts (e.g. <code>ovm_report_info</code> ).
<code>`ovm_*_imp_decl</code>	OK to use. These enable a component to implement more than one instance of a TLM interface. Non-macro solutions don't provide significant advantage.
<code>`ovm_field_*</code>	Do not use. These inject lots of complex code that substantially decreases performance and hinders debug.
<code>`ovm_do_*</code>	Do not use. These unnecessarily obscure a simple API and are best replaced by a user-defined task, which affords far more flexibility.
<code>`ovm_sequence_utils</code> <code>`ovm_sequencer_utils</code> <b>other related macros</b>	Do not use. These macros built up a sequencer's sequence library and enable automatic starting of sequences, which is almost always the wrong thing to do.

---

# Analysis Components & Techniques

---

## Analysis

---

Components in a UVM testbench that observe and analyze behavior of the DUT

### Analysis Chapter contents:

[Analysis \(this page\)](#)

[AnalysisPort](#)

[AnalysisConnections](#)

[MonitorComponent](#)

[Predictors](#)

[Scoreboards](#)

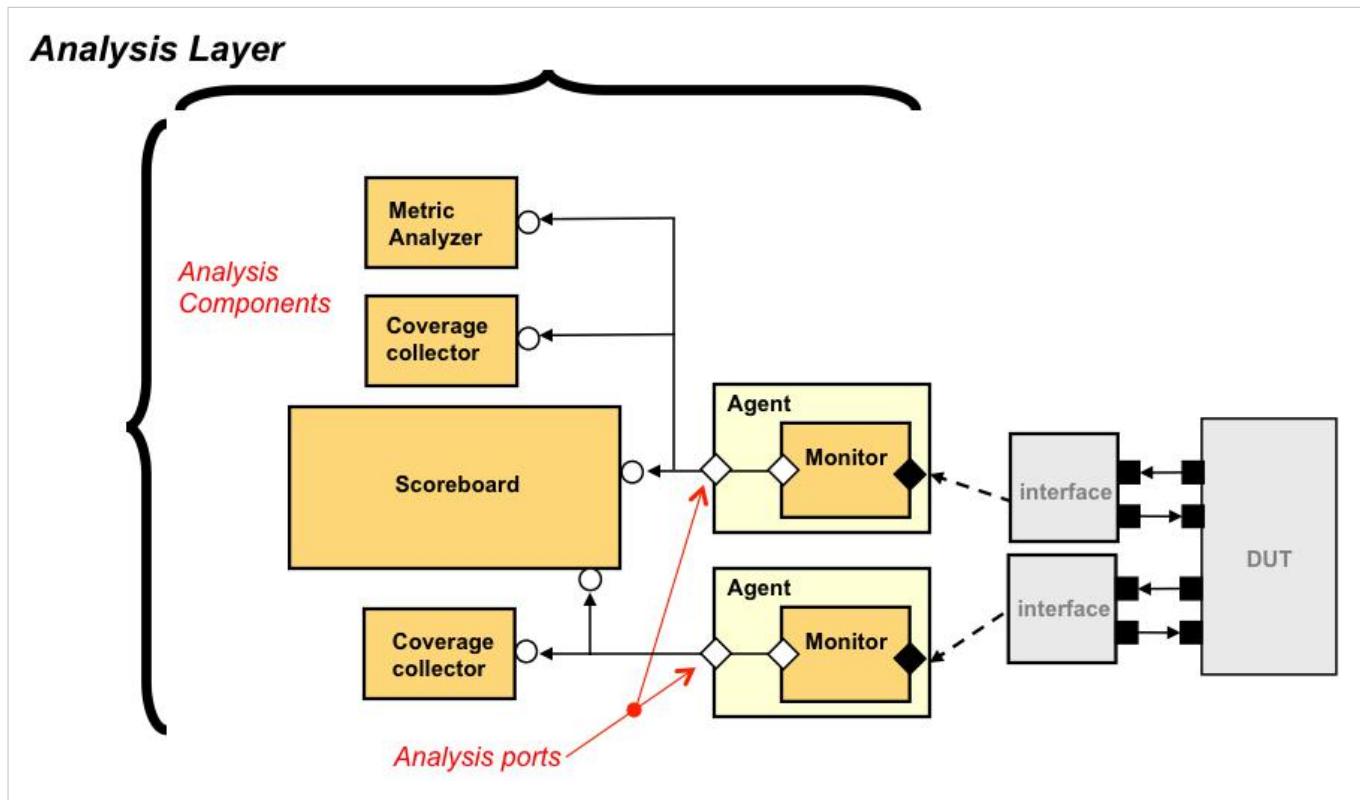
[MetricAnalyzers](#)

[Matlab/Integration](#)

## Topic Overview

Verifying a design consists of two major parts: stimulus generation and an analysis of the designs response. Stimulus generation sets up the device and puts it in a particular state, then the analysis part actually performs the verification.

The analysis portion of a testbench is made up of components that observe behavior and make a judgement whether or not the device conforms to its specification. Examples of specified behavior include functional behavior, performance, and power utilization.



## Monitoring DUT Activity

The process by which the analysis section makes its judgement starts with observing response activity in the device under test (DUT). This is done by one or more monitors that observe the signal-level activity on the DUT through a virtual interface(s). The monitor converts signal-level activity into TLM transactions, and broadcasts the transactions to interested analysis components using analysis ports which are connected to subscribers. These subscribers capture the transactions and perform their analysis.

## Scoreboards

These analysis components collect the transactions sent by the monitor and perform specific analysis activities on the collection of transactions. Scoreboard components determine whether or not the device is functioning properly. The best scoreboard architecture separates its tasks into two areas of concern: prediction and evaluation.

A predictor model, sometimes referred to as a "Golden Reference Model", receives the same stimulus stream as the DUT and produces known good response transaction streams. The scoreboard evaluates the predicted activity with actual observed activity on the DUT.

A common evaluation technique when there is one expected stream and one actual stream is to use a comparator, which can either compare the transactions assuming in-order arrival of transactions or out-of-order arrival.

## Coverage Objects

Coverage information is used to answer the questions "Are we done testing yet?" and "Have we done adequate testing?".

Coverage objects are collectors which subscribe to a monitor's analysis ports and sample observed activity into SystemVerilog functional coverage constructs. The data from each test is entered into a shared coverage database, which is used to determine overall verification progress.

## Metric Analyzers

Metric Analyzers watch and record non-functional behavior such as timing/performance and power utilization. Their architecture is generally standard. Depending on the number of transaction streams they observe, they are implemented as an uvm\_subscriber or with analysis exports. They can perform ongoing calculations during the run phase, and/or during the post-run phases.

## Analysis Reporting

All data is collected during simulation. Most analysis takes place dynamically as the simulation runs, but some analysis can be done after the run phase ends. UVM provides three phases for this purpose: extract\_phase, check\_phase, and report\_phase. These phases allow components to optionally extract relevant data collected during the run, perform a check, and then finally produce reports about all the analysis performed, whether during the run or post-run.

# AnalysisPort

## Overview

One of the unique aspects of the analysis section of a testbench is that usually there are many independent calculations and evaluations all operating on the same piece of data. Rather than lump all these evaluations into a single place, it is better to separate them into independent, concurrent components. This leads to a topological pattern for connecting components that is common to the analysis section: the one-to-many topology, where one connection point broadcasts information to many connected components that read and analyze the data.

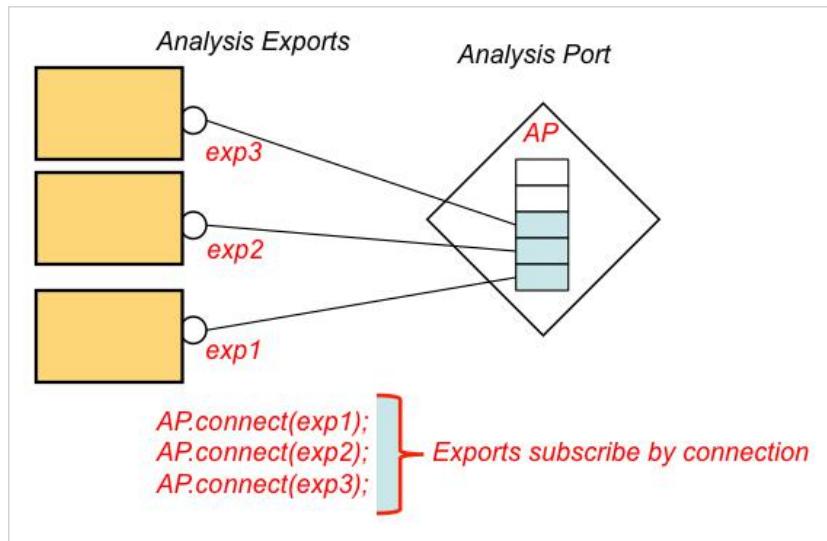
This connection topology implementation behavior lends itself to an OOP design pattern called the "observer pattern". In this pattern, interested "observers" register themselves with a single information source. There is no minimum or maximum number of observers (e.g. the number of observers could be zero). The information source performs a single operation to broadcast the data to all registered observers.

An additional requirement of UVM Analysis is "Do not interfere with the DUT". This means that the act of broadcasting must be a non-blocking operation.

UVM provides three objects to meet the requirements of the observer pattern as well as the non-interference requirement: analysis ports, analysis exports, and analysis fifos.

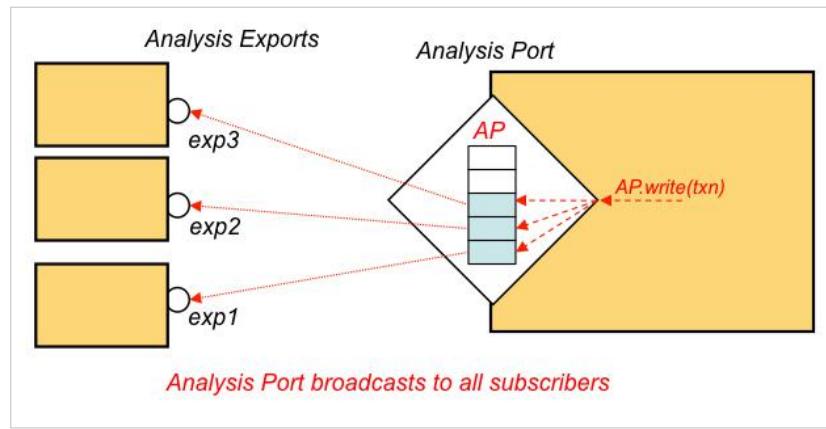
## Detail

Analysis ports, analysis exports, and analysis fifos follow the standard UVM transaction-level communication semantics. An analysis port **requires** an implementation of write() to be connected to it. An analysis export **provides** the implementation of the write() function. As with other UVM TLM ports, analysis ports are parameterized classes where the parameter is the transaction type being passed. Ports provide a local object through which code can call a function. Exports are connection points on components that provide the implementation of the functions called through the ports. Ports and exports are connected through a call to the connect() function.



All other UVM TLM ports and exports, such as blocking put ports and blocking put exports, perform point-to-point communication. Because of the one-to-many requirement for analysis ports and exports, an analysis port allows multiple analysis exports to be connected to it. It also allows for no connections to be present. The port maintains a list of connected exports.

The Analysis port provides a single void function named write() to perform the broadcast behavior. When code calls the write() function on an analysis port, the port then uses its internal list to broadcast by calling write() on all connected exports. This causes the write() function to be executed on all components containing the connected exports. If no exports are connected to the analysis port, then no action is performed when the write() function is called and no error occurs.

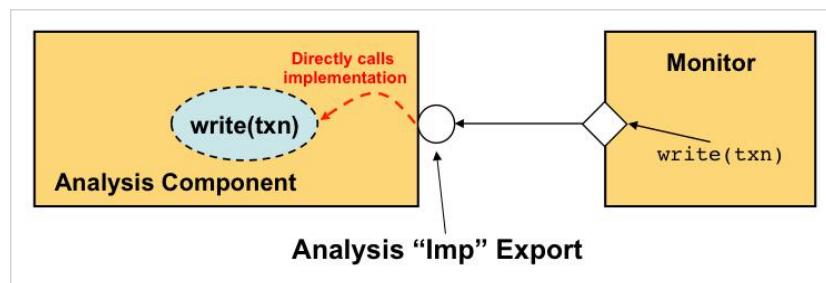


Analysis ports adhere to the non-interference requirement by providing the write() operation as a function, rather than a task. Because it is a function, it cannot block.

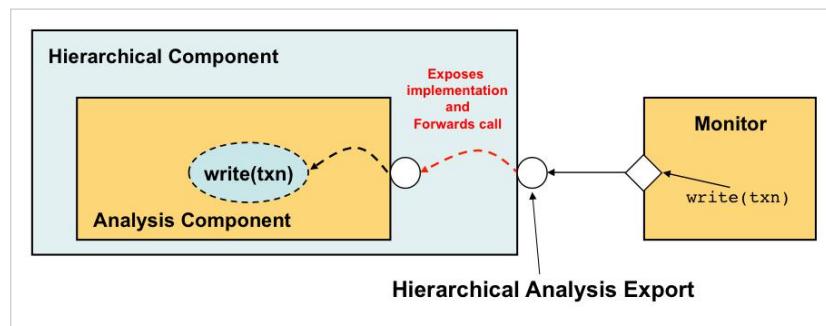
## Analysis Exports

You register an analysis export as an observer with the analysis port by passing the export as the argument to the port's connect() function. As with other TLM exports, an analysis export comes in two types: a hierarchical export or an "imp" export. Both hierarchical and "imp" exports can be connected to a port.

An "imp" export is placed on a component that actually implements the write() function directly.

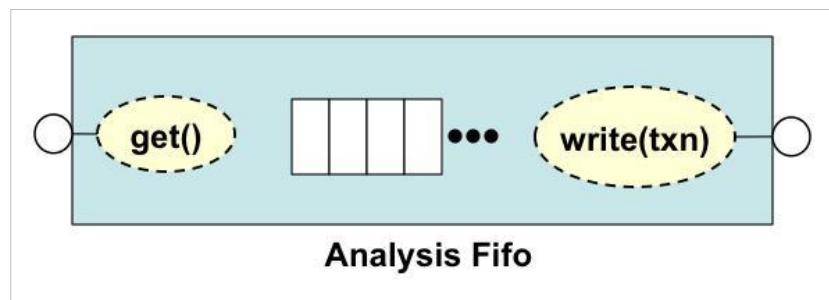


A Hierarchical export is used for hierarchical connections, where the component that implements the write() function is a hierarchical child of the component with the export. A hierarchical export forwards the call to write() to the child component.

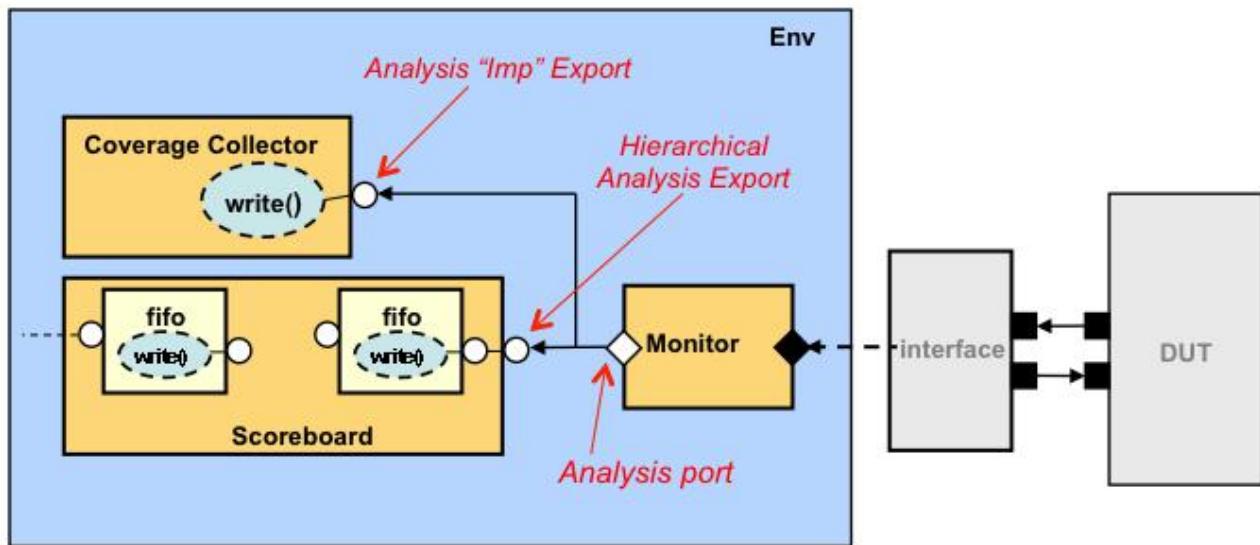


## Analysis Fifos

UVM provides a pre-built component called an analysis fifo that has an "imp"-style export and an implementation of write() that places the data written into a fifo buffer. The buffer can grow indefinitely, in order to prevent blocking and adhere to the non-interference requirement. The analysis fifo extends the tlm\_fifo class, so it also has all of the exports and operations of a tlm fifo such as blocking get, etc.



## Implementing the write() function



The analysis component is required to implement the write() function that is called by the Monitor's analysis port. For an analysis component that has a single input stream, you can extend the uvm\_subscriber class. For components that have multiple input streams, you can either directly implement "write()" functions and provide an "imp" exports, or you can expose hierarchical children's implementations of write() by providing hierarchical exports. The decision of which to use depends on the component's functionality and your preferred style of coding.

In the diagram above, the Coverage Collector extends the uvm\_subscriber class, which has an analysis "imp" export. The extended class then implements the write() function.

The Scoreboard has two input streams of transactions and so uses embedded Analysis fifos which buffer the incoming streams of transactions. In this case the write() method is implemented in the fifos, so the Scoreboard uses hierarchical analysis exports to expose the write() method externally.

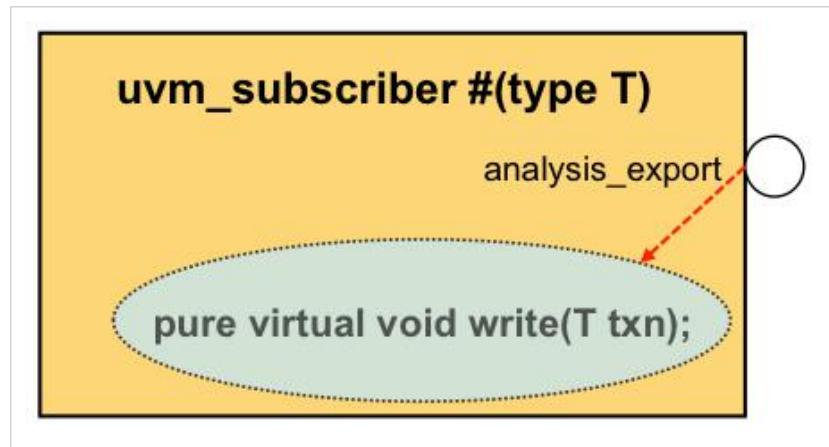
# AnalysisConnections

## Overview

An analysis component captures transaction data by implementing the write() function that is called by the analysis port to which it is connected. Depending on the functionality of the analysis component, you either directly implement the write() function and provide an "imp" export using the uvm\_analysis\_imp class, or you expose a hierarchical child's implementation of write() by providing a hierarchical export using the uvm\_analysis\_export class.

### Single Transaction Stream: uvm\_subscriber

Many analysis components just need to sample the data sent by the Monitor, and perform some calculation such as coverage measurement. For these components, which only deal with a single stream of transactions, UVM provides a base component class called uvm\_subscriber. This class extends uvm\_component and includes a single "imp"-style export named analysis\_export, with a pure virtual declaration of write(). To use this class, you just extend it and override the write() function to perform the analysis calculation.



```
class coverage_sb extends uvm_subscriber #(Packet);
  `uvm_component_utils(coverage_sb)

  Packet pkt;
  int pkt_cnt;

  covergroup cov1;
    s: coverpoint pkt.src_id {
      bins src[8] = {[0:7]};
    }
    d: coverpoint pkt.dest_id {
      bins dest[8] = {[0:7]};
    }
    cross s, d;
  endgroup : cov1

  function new( string name , uvm_component parent);
    super.new( name , parent );
    cov1 = new();
  endfunction
```

```

function void write(Packet t);
    real current_coverage;
    pkt = t;
    pkt_cnt++;
    cov1.sample(); // cause sampling of covergroup
    current_coverage = $get_coverage();
    uvm_report_info("COVERAGE", $psprintf("%0d Packets sampled, Coverage = %f%% ",
                                         pkt_cnt, current_coverage));
endfunction

endclass

```

## Two styles of implementation

There are two approaches you can take to implement the behavior of an analysis component with more than one input stream. The approaches are essentially equivalent, in that any implementation using one approach can be equivalently implemented using the other. Most of the time it just boils down to a personal preference.

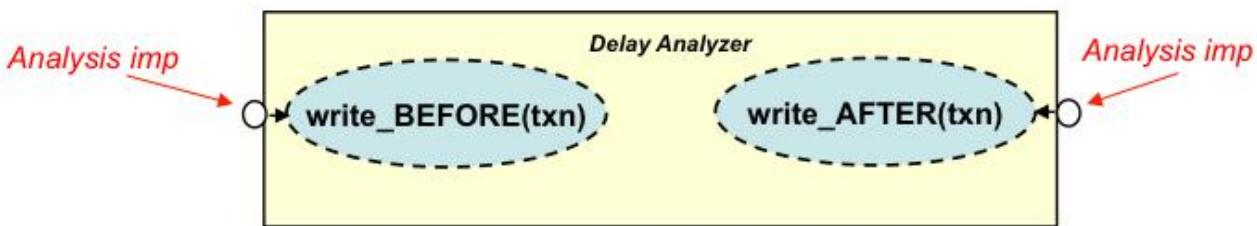
In one approach, you implement all the write() functions directly, and declare imp connections in the containing component. Because the write() functions must be non-blocking, you cannot directly perform any synchronization with other streams. If synchronization is required, you must deposit the transaction in some kind of storage element (e.g. a fifo or associative array) and provide a separate process, usually in the run() task, to perform the synchronization.

In the other approach, you provide an embedded analysis fifo for each input stream, and declare a hierarchical analysis export to expose the fifo externally. In this case, the write() functions are provided by the fifos and they deposit the transactions into the fifo data structures. You then implement the functionality of the analysis component through the run() task. Using this approach involves an extra step of connecting the analysis fifos to the analysis exports.

## Multiple Transaction Streams Without Synchronization

When the analysis component needs to sample multiple transaction streams, it adds complications to writing the behavior of the component.

One complication when using uvm\_analysis\_imps with multiple streams is that each uvm\_analysis\_imp expects to be tied to a write() function. Since there can only be one function named write() in the component, a workaround is needed. UVM provides a workaround by defining the `uvm\_analysis\_imp\_decl macro. This macro allows you to declare a specialized "imp"-style analysis export that calls a function named write\_SUFFIX where you specify \_SUFFIX.



```

// Declare the suffixes that will be appended to the imps and functions
`uvm_analysis_imp_decl(_BEFORE)
`uvm_analysis_imp_decl(_AFTER)

```

```
class delay_analyzer extends uvm_component;
`uvm_component_utils(delay_analyzer)

// Declare the imps using the suffixes declared above.
// The first parameter is the transaction type.
// The second parameter is the type of component that implements the interface
// Usually, the second parameter is the same as the containing component type
  uvm_analysis_imp_BEFORE #(alu_txn, delay_analyzer) before_export;
  uvm_analysis_imp_AFTER #(alu_txn, delay_analyzer) after_export;

  real m_before[$];
  real m_after[$];
  real last_b_time, last_a_time;
  real longest_b_delay, longest_a_delay;

  function new( string name , uvm_component parent ) ;
    super.new( name , parent );
    last_b_time = 0.0;
    last_a_time = 0.0;
  endfunction

  // Implement the interface function by appending the function name with
  // the suffix declared in the macro above.
  function void write_BEFORE(alu_txn t);
    real delay;
    delay = $realtime - last_b_time;
    last_b_time = $realtime;
    m_before.push_back(delay);
  endfunction

  // Implement the interface function by appending the function name with
  // the suffix declared in the macro above.
  function void write_AFTER(alu_txn t);
    real delay;
    delay = $realtime - last_a_time;
    last_a_time = $realtime;
    m_after.push_back(delay);
  endfunction

  function void build_phase( uvm_phase phase );
    // The second argument to the imp constructor is a handle to the object
    // that implements the interface functions.  It should be of the type
    // specified in the declaration of the imp.  Usually, it is "this".
  endfunction
```

```
before_export = new("before_export", this);
after_export = new("after_export", this);
endfunction

function void extract_phase( uvm_phase phase );
foreach (m_before[i])
  if (m_before[i] > longest_b_delay) longest_b_delay = m_before[i];
foreach (m_after[i])
  if (m_after[i] > longest_a_delay) longest_a_delay = m_after[i];
endfunction

function void check_phase( uvm_phase phase );
string s;
if (longest_a_delay > 100.0) begin
  $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
  uvm_report_warning("Delay Analyzer",s);
end
if (longest_b_delay > 100.0) begin
  $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
  uvm_report_warning("Delay Analyzer",s);
end
endfunction

function void report_phase( uvm_phase phase );
uvm_report_info("Delay Analyzer", $sformatf("Longest BEFORE delay:      %5.2f", longest_b_delay));
uvm_report_info("Delay Analyzer", $sformatf("Longest AFTER delay:      %5.2f", longest_a_delay));
endfunction

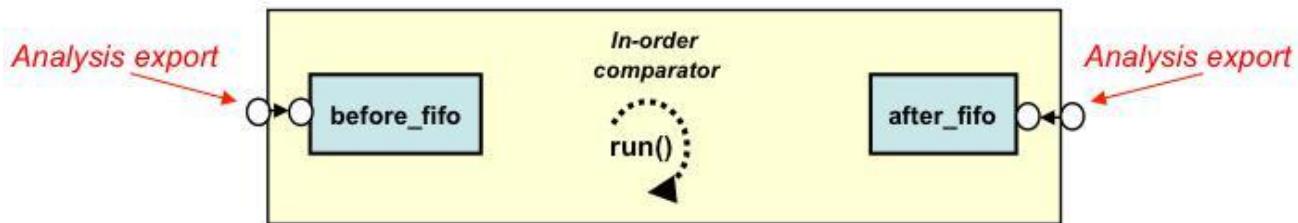
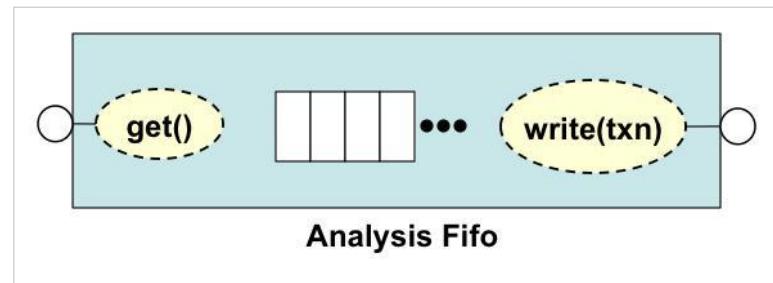
endclass
```

## Multiple Transaction Streams With Synchronization

Implementing the write() functions directly is easier when the required behavior of the analysis component allows for each stream's processing to be done independently. Many situations, however, require synchronization between the transaction streams, so that data is not lost. For example, an in-order Comparator has two streams, one from the Predictor and one from the Monitor. It must wait until it has a matching pair of transactions, one from each stream, to compare.

In this case, rather than performing all the synchronization work manually in two write() functions, you can instantiate two analysis\_fifos and place two hierarchical exports on the Comparator. Since you are not implementing the behavior of the Comparator in the write() functions, a different approach is required. In this case, you can implement the

Comparator behavior in the run() task of the component and use the blocking operations on the analysis fifos to perform the necessary synchronization.



```

class comparator_inorder extends uvm_component;
`uvm_component_utils(comparator_inorder)

uvm_analysis_export #(alu_txn) before_export;
uvm_analysis_export #(alu_txn) after_export;

uvm_tlm_analysis_fifo #(alu_txn) before_fifo, after_fifo;
int m_matches, m_mismatches;

function new( string name , uvm_component parent ) ;
super.new( name , parent );
m_matches = 0;
m_mismatches = 0;
endfunction

function void build_phase( uvm_phase phase );
before_fifo = new("before_fifo", this);
after_fifo = new("after_fifo", this);
before_export = new("before_export", this);
after_export = new("after_export", this);
endfunction

function void connect_phase( uvm_phase phase );
before_export.connect(before_fifo.analysis_export);
after_export.connect(after_fifo.analysis_export);
endfunction

```

```
task run_phase( uvm_phase phase );
  string s;
  alu_txn before_txn, after_txn;
  forever begin
    before_fifo.get(before_txn);
    after_fifo.get(after_txn);
    if (!before_txn.compare(after_txn)) begin
      $sformat(s, "%s does not match %s", before_txn.convert2string(), after_txn.convert2string());
      uvm_report_error("Comparator Mismatch",s);
      m_mismatches++;
    end else begin
      m_matches++;
    end
  end
endtask

function void report_phase( uvm_phase phase );
  uvm_report_info("Inorder Comparator", $sformatf("Matches:      %0d", m_matches));
  uvm_report_info("Inorder Comparator", $sformatf("Mismatches: %0d", m_mismatches));
endfunction

endclass
```

For more complicated synchronization needs, you would use a combination of multiple write\_SUFFIX() functions which would place transaction data into some kind of shared data structure, along with code in the run() task to perform coordination and control.

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# MonitorComponent

## Overview

The first task of the analysis portion of the testbench is to monitor activity on the DUT. A Monitor, like a Driver, is a constituent of an agent. A monitor component is similar to a driver component in that they both perform a translation between actual signal activity and an abstract representation of that activity. The key difference between a Monitor and a Driver is that a Monitor is always passive. It does not drive any signals on the interface. When an agent is placed in passive mode, the Monitor continues to execute.

A Monitor communicates with DUT signals through a virtual interface, and contains code that recognizes protocol patterns in the signal activity. Once a protocol pattern is recognized, a Monitor builds an abstract transaction model representing that activity, and broadcasts the transaction to any interested components.

## Construction

Monitors should extend `uvm_monitor`. They should have one analysis port and a virtual interface handle that points to a DUT interface.

```
class wb_bus_monitor extends uvm_monitor;
  `uvm_component_utils(wb_bus_monitor)

  uvm_analysis_port #(wb_txn) wb_mon_ap;
  virtual wishbone_bus_syscon_if m_v_wb_bus_if;
  wb_config m_config;

  // Standard component constructor
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase( uvm_phase phase );
    super.build_phase( phase );
    wb_mon_ap = new("wb_mon_ap", this);
    m_config = wb_config::get_config(this); // get config object
  endfunction

  function void connect_phase( uvm_phase phase );
    super.connect_phase( phase );
    m_v_wb_bus_if = m_config.v_wb_bus_if; // set local virtual if property
  endfunction

  // run_phase not shown...

endclass
```

## Passive Monitoring

Like in a scientific experiment, where the act of observing should not affect the activity observed, monitor components should be passive. They should not inject any activity into the DUT signals. Practically speaking, this means that monitor code should be completely read-only when interacting with DUT signals.

## Recognizing Protocols

A monitor must have knowledge of protocol in order to detect recognizable patterns in signal activity. Detection can be done by writing protocol-specific state machine code in the monitor's run() task. This code waits for a pattern of key signal activity by observing through the virtual interface handle.

## Building Transaction Objects

Once the pattern is recognized, monitors build one or possibly more transactions that abstractly represents the signal activity. This can be done by calling a function and passing in the transaction-specific attributes (e.g. data value and address value) as arguments to the function, or by setting transaction attributes on an existing transaction as they are detected.

## Copy-on-Write Policy

Since objects in SystemVerilog are handle-based, when a Monitor writes a transaction handle out of its analysis port, only the handle gets copied and broadcast to subscribers. This write operation happens each time the Monitor runs through its ongoing loop of protocol recognition in the run() task. To prevent overwriting the same object memory in the next iteration of the loop, the handle that is broadcast should point to a separate copy of the transaction object that the Monitor created.

This can be accomplished in two ways:

- Create a new transaction object in each iteration of (i.e. inside) the loop
- Reuse the same transaction object in each iteration of the loop, but clone the object immediately prior to calling write() and broadcast the handle of the clone.

## Broadcasting the Transaction

Once a new or cloned transaction has been built it should be broadcast to all interested observers by writing to an analysis port.

## Example Monitor

```
task run_phase( uvm_phase phase );
    wb_txn txn, txn_clone;
    txn = wb_txn::type_id::create("txn"); // Create once and reuse
    forever @ (posedge m_v_wb_bus_if.clk)
        if(m_v_wb_bus_if.s_cyc) begin // Is there a valid wb cycle?
            txn.adr = m_v_wb_bus_if.s_addr; // get address
            txn.count = 1; // set count to one read or write
            if(m_v_wb_bus_if.s_we) begin // is it a write?
                txn.data[0] = m_v_wb_bus_if.s_wdata; // get data
                txn.txn_type = WRITE; // set op type
            end
        end
    endforever
endtask
```

```
while (! (m_v_wb_bus_if.s_ack[0] | m_v_wb_bus_if.s_ack[1] | m_v_wb_bus_if.s_ack[2]))
  @ (posedge m_v_wb_bus_if.clk); // wait for cycle to end
end
else begin
  txn.txn_type = READ; // set op type
  case (1) //Nope its a read, get data from correct slave
    m_v_wb_bus_if.s_stb[0]: begin
      while (! (m_v_wb_bus_if.s_ack[0])) @ (posedge m_v_wb_bus_if.clk); // wait for ack
      txn.data[0] = m_v_wb_bus_if.s_rdata[0]; // get data
    end
    m_v_wb_bus_if.s_stb[1]: begin
      while (! (m_v_wb_bus_if.s_ack[1])) @ (posedge m_v_wb_bus_if.clk); // wait for ack
      txn.data[0] = m_v_wb_bus_if.s_rdata[1]; // get data
    end
  endcase
end
$cast(txn_clone, txn.clone()); // Clone txn to protect from next loop iteration overwriting
wb_mon_ap.write(txn_clone); // broadcast the cloned txn
end
endtask
```

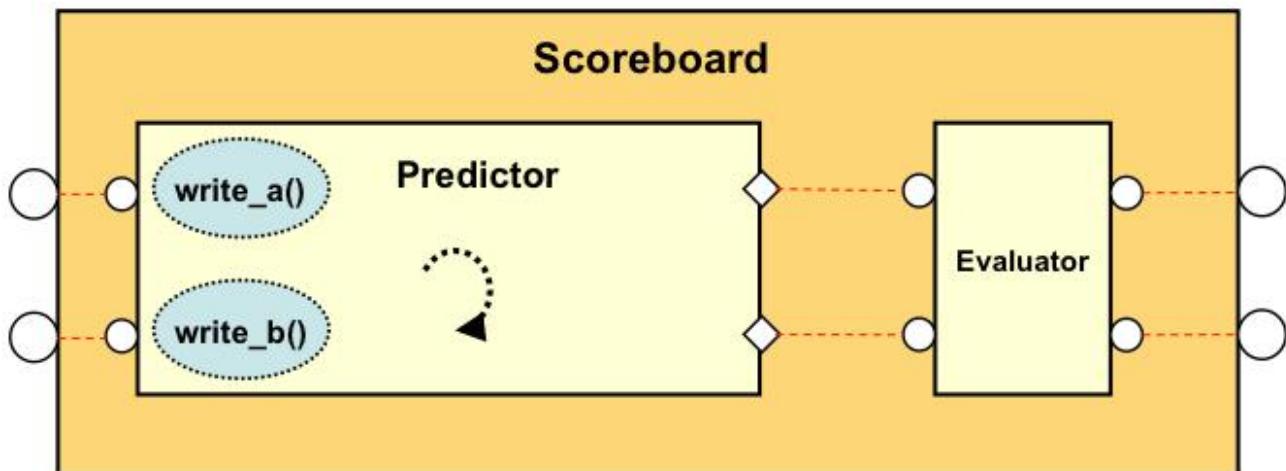
( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

# Predictors

## Overview

A Predictor is a verification component that represents a "golden" model of all or part of the DUT functionality. It takes the same input stimulus that is sent to the DUT and produces expected response data that is by definition correct.

## Predictors in the Testbench Environment



Predictors are a part of the Scoreboard component that generate expected transactions. They should be separate from the part of the scoreboard that performs the evaluation. A predictor can have one or more input streams, which are the same input streams that are applied to the DUT.

## Construction

Predictors are typical analysis components that are subscribers to transaction streams. The inputs to a predictor are transactions generated from monitors observing the input interfaces of the DUT. The predictors take the input transaction(s) and process them to produce expected output transactions. Those output transactions are broadcast through analysis ports to the evaluator part of the scoreboard, and to any other analysis component that needs to observe predicted transactions. Internally, predictors can be written in C, C++, SV or SystemC, and are written at an abstract level of modeling. Since predictors are written at the transaction level, they can be readily chained if needed.

## Example

```
class alu_tlm extends uvm_subscriber #(alu_txn);
  `uvm_component_utils(alu_tlm)

  uvm_analysis_port #(alu_txn) results_ap;

  function new(string name, uvm_component parent );
    super.new( name , parent );
  endfunction

  function void build_phase( uvm_phase phase );
  endfunction
```

```
results_ap = new("results_ap", this);
endfunction

function void write( alu_txn t);
  alu_txn out_txn;
  $cast(out_txn,t.clone());
  case(t.mode)
    ADD: out_txn.result = t.val1 + t.val2;
    SUB: out_txn.result = t.val1 - t.val2;
    MUL: out_txn.result = t.val1 * t.val2;
    DIV: out_txn.result = t.val1 / t.val2;
  endcase
  results_ap.write(out_txn);
endfunction

endclass
```

## Predictor as a Proxy for the DUT

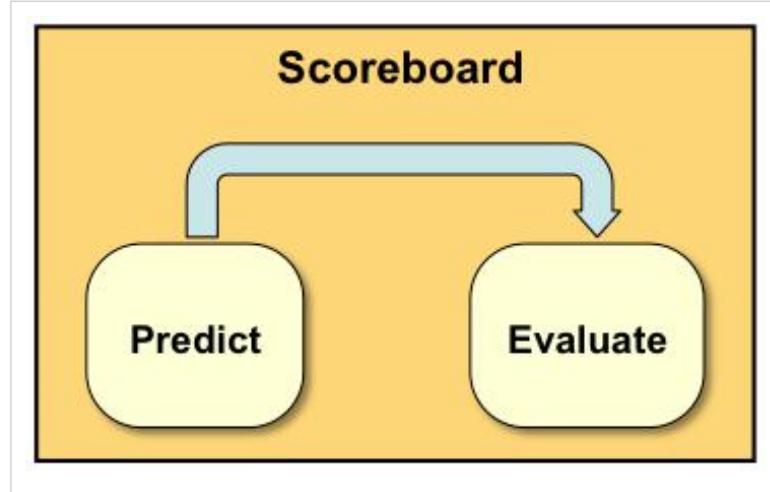
Another use of a Predictor is to act as a proxy DUT while the DUT is being written. Typically, since the Predictor is written at a higher level of abstraction, it takes less time to write and is available earlier than the DUT. As a proxy for the DUT, it allows testbench development and debugging to proceed in parallel with DUT development.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Scoreboards

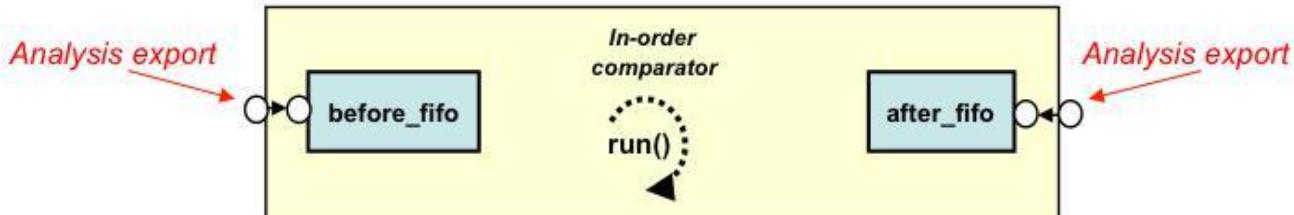
## Overview

The Scoreboard's job is to determine whether or not the DUT is functioning properly. The scoreboard is usually the most difficult part of the testbench to write, but it can be generalized into two parts: The first step is determining what exactly is the correct functionality. Once the correct functionality is predicted, the scoreboard can then evaluate whether or not the actual results observed on the DUT match the predicted results. The best scoreboard architecture is to separate the prediction task from the evaluation task. This gives you the most flexibility for reuse by allowing for substitution of predictor and evaluation models, and follows the best practice of separation of concerns.



In cases where there is a single stream of predicted transactions and a single stream of actual transactions, the scoreboard can perform the evaluation with a simple comparator. The most common comparators are an in-order and out-of-order comparator.

## Comparing Transactions Assuming In-Order Arrival



An in-order comparator assumes that matching transactions will appear in the same order from both expected and actual streams. It gets transactions from the expected and actual side and evaluates them. The transactions will arrive independently, so the evaluation must block until both transactions are present. In this case, an easy implementation would be to embed two analysis fifos in the comparator and perform the synchronization and evaluation in the run() task. Evaluation can be as simple as calling the transaction's compare() method, or it can be more involved, because for the purposes of evaluating correct behavior, comparison does not necessarily mean equality.

```
class comparator_inorder extends uvm_component;
`uvm_component_utils(comparator_inorder)

uvm_analysis_export #(alu_txn) before_export;
uvm_analysis_export #(alu_txn) after_export;

uvm_tlm_analysis_fifo #(alu_txn) before_fifo, after_fifo;
```

```
int m_matches, m_mismatches;

function new( string name , uvm_component parent) ;
  super.new( name , parent );
  m_matches = 0;
  m_mismatches = 0;
endfunction

function void build_phase( uvm_phase phase );
  before_fifo = new("before_fifo", this);
  after_fifo = new("after_fifo", this);
  before_export = new("before_export", this);
  after_export = new("after_export", this);
endfunction

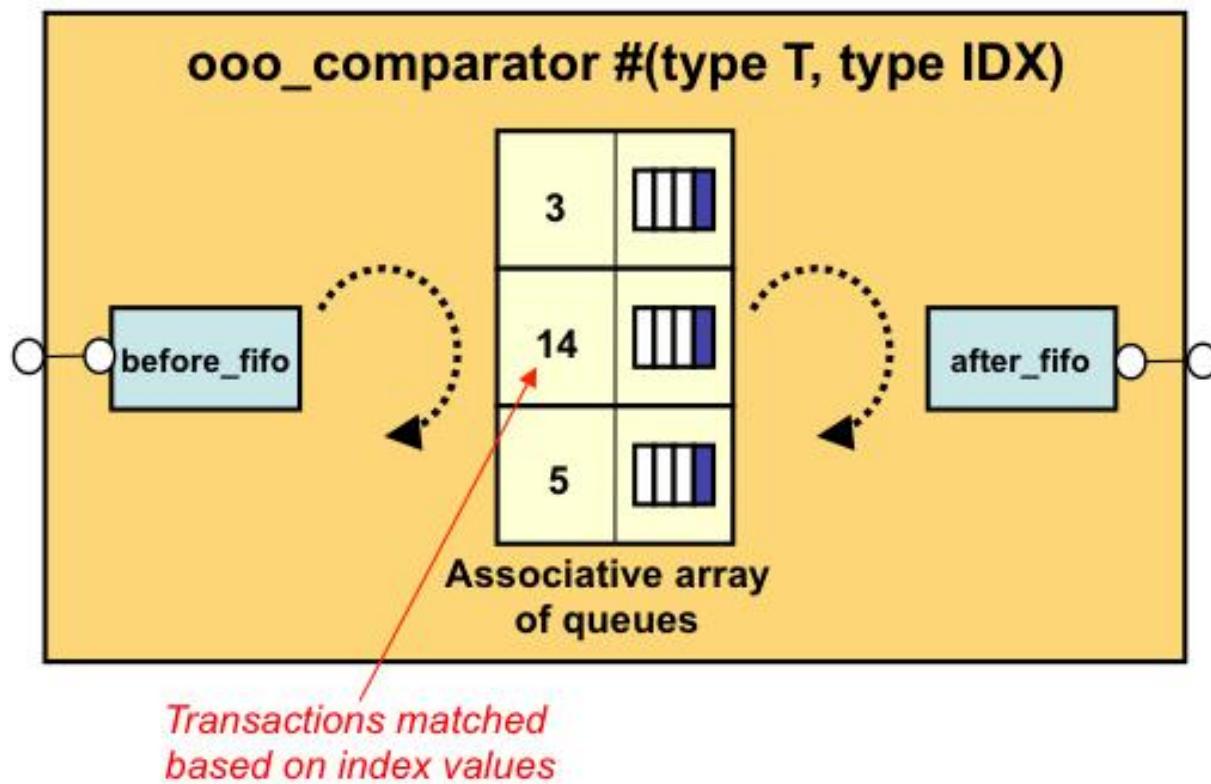
function void connect_phase( uvm_phase phase );
  before_export.connect(before_fifo.analysis_export);
  after_export.connect(after_fifo.analysis_export);
endfunction

task run_phase( uvm_phase phase );
  string s;
  alu_txn before_txn, after_txn;
  forever begin
    before_fifo.get(before_txn);
    after_fifo.get(after_txn);
    if (!before_txn.compare(after_txn)) begin
      $sformat(s, "%s does not match %s", before_txn.convert2string(), after_txn.convert2string());
      uvm_report_error("Comparator Mismatch",s);
      m_mismatches++;
    end else begin
      m_matches++;
    end
  end
endtask

function void report_phase( uvm_phase phase );
  uvm_report_info("Inorder Comparator", $sformatf("Matches:      %0d", m_matches));
  uvm_report_info("Inorder Comparator", $sformatf("Mismatches: %0d", m_mismatches));
endfunction

endclass
```

## Comparing transactions out-of-order



An out-of-order comparator makes no assumption that matching transactions will appear in the same order from the expected and actual sides. So, unmatched transactions need to be stored until a matching transaction appears on the opposite stream. For most out-of-order comparators, an associative array is used for storage. This example comparator has two input streams arriving through analysis exports. The implementation of the comparator is symmetrical, so the export names do not have any real importance. This example uses embedded fifos to implement the analysis write() functions, but since the transactions are either stored into the associative array or evaluated upon arrival, this example could easily be written using analysis imps and write() functions.

Because of the need to determine if two transactions are a match and should be compared, this example requires transactions to implement an index\_id() function that returns a value that is used as a key for the associative array. If an entry with this key already exists in the associative array, it means that a transaction previously arrived from the other stream, and the transactions are compared. If no key exists, then this transaction is added to associative array.

This example has an additional feature in that it does not assume that the index\_id() values are always unique on a given stream. In the case where multiple outstanding transactions from the same stream have the same index value, they are stored in a queue, and the queue is the value portion of the associative array. When matches from the other stream arrive, they are compared in FIFO order.

```
class ooo_comparator
  #(type T = int,
  type IDX = int)
  extends uvm_component;

  typedef ooo_comparator #(T, IDX) this_type;
  `uvm_component_param_utils(this_type)
```

```
typedef T q_of_T[$];
typedef IDX q_of_IDX[$];

uvm_analysis_export #(T) before_axp, after_axp;

protected uvm_tlm_analysis_fifo #(T) before_fifo, after_fifo;
bit before_queued = 0;
bit after_queued = 0;

protected int m_matches, m_mismatches;

protected q_of_T received_data[IDX];
protected int rcv_count[IDX];

protected process before_proc = null;
protected process after_proc = null;

function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction

function void build_phase( uvm_phase phase );
    before_axp = new("before_axp", this);
    after_axp = new("after_axp", this);
    before_fifo = new("before", this);
    after_fifo = new("after", this);
endfunction

function void connect_phase( uvm_phase phase );
    before_axp.connect(before_fifo.analysis_export);
    after_axp.connect(after_fifo.analysis_export);
endfunction : connect

// The component forks two concurrent instantiations of this task
// Each instantiation monitors an input analysis fifo
protected task get_data(ref uvm_tlm_analysis_fifo #(T) txn_fifo, ref process proc, input bit is_before);
    T txn_data, txn_existing;
    IDX idx;
    string rs;
    q_of_T tmpq;
    bit need_to_compare;
    forever begin
        proc = process::self();
        if (is_before)
            before_axp.read(txn_data);
        else
            after_axp.read(txn_data);
        if (tx

```

```
// Get the transaction object, block if no transaction available
txn_fifo.get(txn_data);
idx = txn_data.index_id();

// Check to see if there is an existing object to compare
need_to_compare = (rcv_count.exists(idx) &&
                    ((is_before && rcv_count[idx] > 0) ||
                     (!is_before && rcv_count[idx] < 0)));
if (need_to_compare) begin
  // Compare objects using compare() method of transaction
  tmpq = received_data[idx];
  txn_existing = tmpq.pop_front();
  received_data[idx] = tmpq;
  if (txn_data.compare(txn_existing))
    m_matches++;
  else
    m_mismatches++;
end
else begin
  // If no compare happened, add the new entry
  if (received_data.exists(idx))
    tmpq = received_data[idx];
  else
    tmpq = {};
  tmpq.push_back(txn_data);
  received_data[idx] = tmpq;
end

// Update the index count
if (is_before)
  if (rcv_count.exists(idx)) begin
    rcv_count[idx]--;
  end
  else
    rcv_count[idx] = -1;
else
  if (rcv_count.exists(idx)) begin
    rcv_count[idx]++;
  end
  else
    rcv_count[idx] = 1;

// If index count is balanced, remove entry from the arrays
```

```
if (rcv_count[idx] == 0) begin
    received_data.delete(idx);
    rcv_count.delete(idx);
end
end // forever
endtask

virtual function int get_matches();
    return m_matches;
endfunction : get_matches

virtual function int get_mismatches();
    return m_mismatches;
endfunction : get_mismatches

virtual function int get_total_missing();
    int num_missing;
    foreach (rcv_count[i]) begin
        num_missing += (rcv_count[i] < 0 ? -rcv_count[i] : rcv_count[i]);
    end
    return num_missing;
endfunction : get_total_missing

virtual function q_of_IDX get_missing_indexes();
    q_of_IDX rv = rcv_count.find_index() with (item != 0);
    return rv;
endfunction : get_missing_indexes;

virtual function int get_missing_index_count(IDX i);
    // If count is < 0, more "before" txns were received
    // If count is > 0, more "after" txns were received
    if (rcv_count.exists(i))
        return rcv_count[i];
    else
        return 0;
endfunction : get_missing_index_count;

task run_phase( uvm_phase phase );
    fork
        get_data(before_fifo, before_proc, 1);
        get_data(after_fifo, after_proc, 0);
    join
endtask : run_phase
```

```
virtual function void kill();
  before_proc.kill();
  after_proc.kill();
endfunction

endclass : ooo_comparator
```

## Advanced Scenarios

In more advanced scenarios, there can be multiple predicted and actual transaction streams coming from multiple DUT interfaces. In this case, a simple comparator is insufficient and the implementation of the evaluation portion of the scoreboard is more complex and DUT-specific.

## Reporting and Recording

The result of the evaluation is a boolean value, which the Scoreboard should use to report and record failures. Usually successful evaluations are not individually reported, but can be recorded for later summary reports.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# MetricAnalyzers

## Overview

Metric Analyzers watch and record non-functional behavior such as latency, power utilization, and other performance-related measurements.

## Construction

Metric Analyzers are generally standard analysis components. They implement their behavior in a way that depends on the number of transaction streams they observe - either by extending `uvm_subscriber` or with analysis imp/exports. They can perform ongoing calculations during the run phase, and/or during the post-run phases.

## Example

```
`uvm_analysis_imp_decl(_BEFORE)
`uvm_analysis_imp_decl(_AFTER)

class delay_analyzer extends uvm_component;
  `uvm_component_utils(delay_analyzer)

  uvm_analysis_imp_BEFORE #(alu_txn, delay_analyzer) before_export;
  uvm_analysis_imp_AFTER #(alu_txn, delay_analyzer) after_export;

  real m_before[$];
  real m_after[$];
  real last_b_time, last_a_time;
  real longest_b_delay, longest_a_delay;

  function new( string name , uvm_component parent) ;
    super.new( name , parent );
    last_b_time = 0.0;
    last_a_time = 0.0;
  endfunction

  // Record when the transaction arrives
  function void write_BEFORE(alu_txn t);
    real delay;
    delay = $realtime - last_b_time;
    last_b_time = $realtime;
    m_before.push_back(delay);
  endfunction

  // Record when the transaction arrives
  function void write_AFTER(alu_txn t);
```

```
real delay;
delay = $realtime - last_a_time;
last_a_time = $realtime;
m_after.push_back(delay);
endfunction

function void build_phase( uvm_phase phase );
  before_export = new("before_export", this);
  after_export = new("after_export", this);
endfunction

// Perform calculation for longest delay metric
function void extract_phase( uvm_phase phase );
  foreach (m_before[i])
    if (m_before[i] > longest_b_delay) longest_b_delay = m_before[i];

  foreach (m_after[i])
    if (m_after[i] > longest_a_delay) longest_a_delay = m_after[i];
endfunction

function void check_phase( uvm_phase phase );
  string s;
  if (longest_a_delay > 100.0) begin
    $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
    uvm_report_warning("Delay Analyzer",s);
  end
  if (longest_b_delay > 100.0) begin
    $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
    uvm_report_warning("Delay Analyzer",s);
  end
endfunction

function void report_phase( uvm_phase phase );
  uvm_report_info("Delay Analyzer", $sformatf("Longest BEFORE delay:      %5.2f", longest_b_delay));
  uvm_report_info("Delay Analyzer", $sformatf("Longest AFTER delay:      %5.2f", longest_a_delay));
endfunction

endclass
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>**  ).

# PostRunPhases

---

## Overview

Many analysis components perform their analysis on an ongoing basis during the simulation run. Sometimes you need to defer analysis until all data has been collected, or a component might need to do a final check at the end of simulation. For these components, UVM provides the post-run phases extract, check, and report.

These phases are executed in a hierarchically bottom-up fashion on all components.

## The Extract Phase

The extract phase allows a component to examine data collected during the simulation run, extract meaningful values and perform arithmetic computation on those values.

## The Check Phase

The check phase allows a component to evaluate any values computed during the extract phase and make a judgment about whether the values are correct. Also, for components that perform analysis continuously during the run, the check phase can be used to check for any missing data or extra data such as unmatched transactions in a scoreboard.

## The Report Phase

The report phase allows a component to display a final report about the analysis in its area of responsibility. A component can be configured whether or not to display its local results, to allow for accumulation and display by higher-level components.

## The Final Phase

UVM adds the Final phase which is normally the very last phase to be executed before simulation \$finish. It is intended that future UVM versions support a multiple test feature which iterates all the preceding phases (e.g. the report phase when complete will jump back to the build phase to execute another test). After all tests are done, the final phase executes once only.

## Example

```
`uvm_analysis_imp_decl(_BEFORE)
`uvm_analysis_imp_decl(_AFTER)

class delay_analyzer extends uvm_component;
`uvm_component_utils(delay_analyzer)

  uvm_analysis_imp_BEFORE #(alu_txn, delay_analyzer) before_export;
  uvm_analysis_imp_AFTER #(alu_txn, delay_analyzer) after_export;

  real m_before[$];
  real m_after[$];
  real last_b_time, last_a_time;
```

```
real longest_b_delay, longest_a_delay;

function new( string name , uvm_component parent) ;
  super.new( name , parent );
  last_b_time = 0.0;
  last_a_time = 0.0;
endfunction

function void write_BEFORE(alu_txn t);
  real delay;
  delay = $realtime - last_b_time;
  last_b_time = $realtime;
  m_before.push_back(delay);
endfunction

function void write_AFTER(alu_txn t);
  real delay;
  delay = $realtime - last_a_time;
  last_a_time = $realtime;
  m_after.push_back(delay);
endfunction

function void build_phase( uvm_phase phase );
  before_export = new("before_export", this);
  after_export = new("after_export", this);
endfunction

function void extract_phase( uvm_phase phase );
  foreach (m_before[i])
    if (m_before[i] > longest_b_delay) longest_b_delay = m_before[i];

  foreach (m_after[i])
    if (m_after[i] > longest_a_delay) longest_a_delay = m_after[i];
endfunction

function void check_phase( uvm_phase phase );
  string s;
  if (longest_a_delay > 100.0) begin
    $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
    uvm_report_warning("Delay Analyzer",s);
  end
  if (longest_b_delay > 100.0) begin
    $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
  end
endfunction
```

```
    uvm_report_warning("Delay Analyzer",s);  
  end  
endfunction  
  
function void report_phase( uvm_phase phase );  
  uvm_report_info("Delay Analyzer", $sformatf("Longest BEFORE delay:      %5.2f", longest_b_delay));  
  uvm_report_info("Delay Analyzer", $sformatf("Longest AFTER delay:      %5.2f", longest_a_delay));  
endfunction  
  
function void final_phase( uvm_phase phase );  
  my_summarize_test_results();  
endfunction  
  
endclass
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm> ).**

# Matlab/Integration

---

## Using MATLAB with UVM/OVM

MATLAB is a modeling tool often used to develop functional models of complex mathematical functions which will then be translated into RTL library blocks. While the translation of the MATLAB function to RTL is a separate topic, the ability to use the source MATLAB code during the verification process can significantly reduce the time to develop a verification testbench. Information about MATLAB and its various capabilities can be found on the Mathworks web site. ([www.mathworks.com](http://www.mathworks.com)<sup>[1]</sup>)

This article expects that the user is familiar with using MATLAB to develop algorithms and understands how MATLAB code is written, organized and run inside a MATLAB environment. Also, separate MATLAB licenses are required to use MATLAB, including licenses for any MATLAB engineering toolboxes or features (specifically EDA Simulator Link) mentioned in this article.

Existing MATLAB functions can be used for both initial Testbench development as a DUT before RTL is available, as well as part of a scoreboard to verify DUT functionality.

MATLAB provides several different mechanisms for using MATLAB functions. Each method has advantages and disadvantages, and which method used should be chosen based upon the specific needs of the user.

MATLAB provides the following capabilities:

- Running MATLAB in parallel with the HDL simulator to perform MATLAB computations.
- Use MATLAB functions as Verilog modules which incorporate MATLAB functions.
- Compiling the MATLAB functions to a shared library allowing functions to be called using the SystemVerilog DPI.
- SystemC TLM2 communication with MATLAB.

The last two methods are somewhat complex and are only briefly discussed below. The example attached with this article only demonstrates the first two capabilities.

### MATLAB in parallel with the HDL simulator

Similar to creating a shared library, there is the ability to run the MATLAB engine in parallel with the HDL simulator. This approach is somewhat easier than creating a shared library in that the MATLAB functions are run directly in the MATLAB engine and doesn't require a separate compilation step. However, the user will need a similar set of DPI functions to start the MATLAB engine, load the MATLAB code and provide the data translation between the simulator and MATLAB engine. Typically, the user will send MATLAB commands to the MATLAB engine using a command string and read the resulting output buffer. The output buffer can then be parsed to read the correct stimulus response and checked inside the scoreboard.

## MATLAB functions as a DUT

MATLAB provides the capability to connect an HDL shell module to a MATLAB function. This capability, called EDA Simulator Link ([www.mathworks.com/products/eda-simulator](http://www.mathworks.com/products/eda-simulator)<sup>[2]</sup>) will run in parallel with the HDL simulator and provide DUT functionality. The inputs and outputs to the MATLAB function are the DUT ports, and any internal functionality is performed by calling the appropriate MATLAB sub-functions

## MATLAB shared libraries

MATLAB provides a compilation tool (mcc) which will compile MATLAB files (.m) files into a shared library. As part of this process, it will create both a shared library file (.so) and header file (.h) to enable using the shared library. To communicate with the shared library, MATLAB requires the use of existing MATLAB matrix functions. To enable simple DPI communication with the MATLAB library, an intermediate wrapper function will need to be developed to provide translation from DPI data types to the MATLAB matrix values.

This coding can be somewhat complex as the translation of DPI variables to the requisite MATLAB matrix values can take some effort. It is somewhat easier to use the MATLAB engine in parallel which provides the ability to call MATLAB functions via MATLAB shell commands instead. This technique is described in the next section.

## MATLAB communication with TLM2 transactions

MATLAB provides a capability to communicate via TLM2 transactions with SystemC environments. Utilizing this method with OVM/UVM would require translation from OVM/UVM <-> SystemC TLM2 <-> MATLAB. Due to the extra SystemC coding requirements, it is not recommended unless the user is already using TLM2 SystemC components.

## MATLAB Integration Example

A complete example is provided below to demonstrate two methods of using MATLAB with a UVM environment. The example consists of three separate use models. The first is a pure UVM environment which uses a Verilog DUT and UVM Testbench with a standard UVM behavioral Scoreboard. The second part replaces the Verilog DUT with a module which uses a MATLAB function as the DUT behavior. The third part replaces the UVM Scoreboard predict\_response() function with the MATLAB function.

## Using MATLAB as a DUT with EDA Simulator Link

For the first stage of our example, we will use a simple 8 bit adder MATLAB function as a DUT. This will be done using a simple MATLAB function and the EDA Simulator Link capability to provide HDL communication between Questa and MATLAB.

### ml\_adder.m

We have created a MATLAB function ml\_adder which takes two integers, adds them and stores the result. It also outputs the result to the MATLAB display so that we can parse the results back into Questa when needed.

The MATLAB source code is shown below.

```
function r = ml_adder(in0, in1)
%MATLAB function to implement a simple adder
```

```
%  
  
r = in0 + in1;  
  
% Print out the results to the MATLAB buffer  
str = sprintf('%d', r);  
disp(str);  
  
% [EOF] ml_adder.m
```

### hdl\_adder.m

To utilize our ml\_adder function as a DUT, we create a MATLAB wrapper function which we can link to an HDL module. The EDA Simulator Link toolbox provides the conversion function between bit vectors and the internal MATLAB data types.

The MATLAB source code is shown below.

```
function [iport,tnext] = hdl_adder(oport, tnow, portinfo)  
% Demonstration for MATLAB Connectivity  
% This function uses the HDL Simulator Link methodology to get HDL port values  
% into and out of this function  
%  
tnext = [];  
iport = struct();  
  
% Convert the input ports to integers  
input0 = mv12dec(oport.input0);  
input1 = mv12dec(oport.input1);  
  
% This is a call to a simple MATLAB function found in ml_adder.m  
output0 = ml_adder(input0,input1);  
  
% Convert and assign the output port to a 9 bit vector  
iport.output0 = dec2mv1(output0,9);  
  
% [EOF] hdl_adder.m
```

## modelsim\_matlab.m

There are two steps to create the link between the Questa and the MATLAB Simulator. First, MATLAB is started and told to create a EDA Simulator Link daemon to listen for communication with Questa. We can use a MATLAB function to do this.

The MATLAB source code is shown below.

```
% This is a MATLAB startup function to create an HDL Simulator Link
% It requires the FLI library to be loaded on the vsim command line
hlddaemon('socket', 5001, 'time', 'int64');

% This adds the MATLAB directory containing the MATLAB functions
addpath ./MATLAB;
```

The second step is to start Questa with the added FLI command to load the MATLAB EDA Simulator Link feature.

MATLAB provides the FLI module used so we add -foreign "matlabclient \$(MATLAB)/toolbox/edalink/extensions/modelsim/linux64/liblfmhdळ\_tmwgcc.so" to the Questa command line.

Once Questa is running, we provide the FLI command to link the testbench DUT module (a shell module in this case), to the MATLAB hdl\_adder function. This is done using the command 'matlabcp /testbench/dut -rising /testbench/dut/valid -mfunc hdl\_adder -socket 5001'. In this example, a script file (scripts/matlab.do) is used to make this easier.

In this example, the testbench (sv/testbench.sv) is compiled with +define+MATLAB. This replaces the existing DUT Verilog module with an empty shell module to connect to MATLAB.

You can run this example with 'make simulate\_matlab\_dut' to see the results. When run, you will see MATLAB console start and the HDL daemon start to listen for connections. After a delay (to give MATLAB time to start), Questa will start and establish the DUT link to MATLAB. When run, you will see the Questa environment run normally as well as see the MATLAB function calculations in the MATLAB console. The output of the MATLAB console is also saved in the file matlab.log if you need to do debugging on the MATLAB functions.

## Using MATLAB as Scoreboard checker

### matlab\_dpi.c

To utilize the MATLAB function as a scoreboard checker, we take a slightly different approach as we now want to call our MATLAB function directly from our SV predictor code instead of communicating via an HDL wrapper. MATLAB provides a C based API which will allow you to start MATLAB as well as send MATLAB commands to the MATLAB engine and read the results back from the output buffer. To simplify these calls, we create a DPI wrapper library to utilize these functions. This DPI module provides 4 basic functions:

int start\_matlab(string cmd) - Starts MATLAB with the given command string. If an empty string is passed, the default 'matlab' command is used. Return value is non-zero if successful.

int send\_matlab\_cmd(string cmd) - Sends a text-based command to MATLAB for evaluation. You can use this to set MATLAB variables, configure MATLAB, or evaluate MATLAB functions. Returns 0 if successful.

string get\_matlab\_buffer() - Gets the MATLAB output buffer from the last MATLAB command sent. You can use this to get the evaluation of the previous MATLAB command sent.

void stop\_matlab() - Causes the MATLAB engine to exit.

matlab\_dpi.c is compiled like any other DPI functions and linked as a shared library. It is linked against two MATLAB libraries (-leng and -lmx) which provide the MATLAB engine functions and MX data communication functions). Also, make sure that you have the MATLAB library directory \$MATLAB/bin/<architecture> in your \$LD\_LIBRARY\_PATH so that the required MATLAB libraries are found.

```
/*
 *      matlab_dpi.c
 *
 *      DPI Functions to enable SV communication to MATLAB
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "engine.h"
#define BUFSIZE 256

Engine *ep;
mxArray *T = NULL, *result = NULL;
char buffer[BUFSIZE+1];

int start_matlab(char *cmd)
{
    if (!ep && !(ep = engOpen(cmd))) {
        fprintf(stderr, "\nCan't start MATLAB engine\n");
        return 0;
    }
    engOutputBuffer(ep, buffer, BUFSIZE);
    return 1;
}

int send_matlab_cmd(char *cmd)
{
    return engEvalString(ep, cmd);
}

char *get_matlab_buffer()
{
    return buffer;
}

void stop_matlab()
{
    engClose(ep);
}
```

## matlab\_dpi\_pkg.svh

To utilize the DPI in our UVM predictor, a SystemVerilog package is created for import when needed.

```
// Package to define MATLAB DPI functions

package matlab_dpi_pkg;

import "DPI-C" function int start_matlab(string cmd);
import "DPI-C" function int send_matlab_cmd(string cmd);
import "DPI-C" function string get_matlab_buffer();
import "DPI-C" function void stop_matlab();

endpackage
```

## predictor\_matlab.svh

Once we have compiled and added the matlab\_dpi.so to the Questa vsim command line, we can start/stop the MATLAB engine and use the MATLAB engine as a scoreboard predictor. The original predictor is extended and we replace the build(), write() and report\_phase() functions to control MATLAB. To use the MATLAB function ml\_adder from above, we simply send the command string "X = ml\_adder(in0, in1)" to MATLAB. MATLAB will call the ml\_adder function and the result is placed in the MATLAB output buffer which we read back and parse the response. The predictor then sends the predicted transaction to the scoreboard predicted analysis port.

```
//-----
//                               Mentor Graphics Corporation
//-----
// Project      : my_project
// Unit        : predictor_matlab
// File        : predictor_matlab.svh
//-----
// Created by   : cgales
// Creation Date : 2012/01/04
//-----
// Title:
// 
// Summary:
// 
// Description:
// 
//-----
// predictor_matlab
//-----
class predictor_matlab extends predictor;
```

```
// factory registration macro
`uvm_component_utils(predictor_matlab)

...
//-----
// build_phase
//-----
function void build_phase(uvm_phase phase);
    // During the build phase, start MATLAB and add
    // MATLAB dir to enable access to MATLAB functions
    if (!start_matlab("matlab -nosplash")) begin
        `uvm_fatal(get_name(), "Unable to start MATLAB");
    end

    void'(send_matlab_cmd("addpath ./MATLAB"));

    m_output_ap = new("m_output_ap", this);
endfunction : build_phase

//-----
// write
//-----
function void write(T t);
    in_tran    m_out_item;
    string msg, cmd, cmd_rsp;

    m_out_item = in_tran::type_id::create("m_out_item");

    //t is the input sequence item (transaction). Process t and then
    // write the new processed output to the m_output_ap.
    m_out_item.do_copy(t);

    $sformat(msg, "INPUT: %s", t.convert2string());
    `uvm_info(get_name(), msg, UVM_HIGH);

    $sformat(cmd, "X = ml_adder(%0d,%0d);", t.input0, t.input1);
    `uvm_info(get_name(), cmd, UVM_HIGH);

    // Call our MATLAB function with our transaction inputs
    void'(send_matlab_cmd(cmd));

    // Readback the MATLAB buffer with our output
    cmd_rsp = get_matlab_buffer();
```

```
`uvm_info(get_name(), $sformatf("MATLAB Buffer is %s", cmd_rsp), UVM_HIGH);

if (!$sscanf(cmd_rsp, ">> %d", m_out_item.output0)) begin
  `uvm_warning(get_name(), "Error parsing MATLAB response");
end

m_output_ap.write(m_out_item);
endfunction : write

function void report_phase(uvm_phase phase);
  // All done - shut down MATLAB
  stop_matlab();
endfunction

endclass : predictor_matlab
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm> ).**

# End Of Test Mechanisms

## EndOfTest

End-of-test Guide for UVM - Learn about the available end-of-test mechanisms and phase objections

### End of Test Chapter contents:

[EndOfTest \(this page\)](#)

[Objections](#)

## Topic Overview

### End of Test in the UVM

A UVM testbench, if is using the standard phasing, has a number of zero time phases to build and connect the testbench, then a number of time consuming phases, and finally a number of zero time cleanup phases.

End of test occurs when all of the time consuming phases have ended. Each phase ends when there are no longer any pending objections to that phase. So end-of-test in the UVM is controlled by managing phase objections. The best way of using phase objections is described in articles linked to from the Phasing Introduction Page.

A simple test might look like this:

```
task reset_phase( uvm_phase phase);
  phase.raise_objection( this );
  reset_seq.start( m_sequencer );
  phase.drop_objection( this );
endtask

task configure_phase( uvm_phase phase);
  phase.raise_objection( this );
  program_control_registers_seq.start( m_sequencer );
  phase.drop_objection( this );
endtask

task main_phase( uvm_phase phase);
  phase.raise_objection( this );
  data_transfer_seq.start( m_sequencer );
  phase.drop_objection( this );
endtask

task shutdown_phase( uvm_phase phase);
  phase.raise_objection( this );

```

```

read_status_registers_seq.start( m_sequencer );
phase.drop_objection( this );
endtask

```

Each of the four phases in the test above raises and drops an objection. Since the particular phases above occur in sequence, then one phase cannot start before the previous one has finished. Raising an objection at the beginning of each phase prevents the phase from immediately terminating, and dropping it means that this component no longer has an objection to the phase ending. The phase will then terminate if there are no other pending objections that have been raised by other components or objects. When there are no pending objections to a particular phase, the simulation will move on the next phase. When there are no time consuming phases left to execute, the simulation moves on to the cleanup phases and the test ends.

## phase\_ready\_to\_end

For sequences, tests, and many complete testbenches, the raising and dropping of phase objections during the normal lifetime of the phase, as described above, is quite sufficient.

However, sometimes a component does not want to actively raise and drop objections during the normal lifetime of a phase, but does want to delay the transition from one phase to the next. This is very often the case in transactors, which for performance reasons cannot raise and drop an objection for every transaction, and is quite often the case for end-to-end scoreboards.

To delay the end of phase after all other components have agreed that the phase should end, that component should raise objections in the phase\_ready\_to\_end method. It is then responsible for dropping those objections, either in the main body of the component or in a task fork / join none'd from the phase\_ready\_end\_method.

An example of using fork / join\_none is shown below :

```

function void my_component::phase_ready_to_end( uvm_phase phase );
  if( !is_ok_to_end() ) begin
    phase.raise_objection( this , "not yet ready to end phase" );
    fork begin
      wait_for_ok_end();
      phase.drop_objection( this , "ok to end phase" );
    end
    join_none
  end
endfunction : phase_ready_to_end

```

Ready\_to\_end\_phase **without** fork / join\_none is used in the Object-to-All and Object-to-One phasing policies often used in components such as transactors and scoreboards.

# Objections

---

## Objections

The uvm\_objection class provides a means for sharing a counter between participating components and sequences. Each participant may "raises" and "drops" objections asynchronously, which increases or decreases the counter value. When the counter reaches zero (from a non-zero value), an "all dropped" condition occurs. The meaning of an all-dropped event depends on the intended application for a particular objection object. For example, the UVM phasing mechanism uses a uvm\_objection object to coordinate the end of each run-time phase. User-processes started in a phase raise and drop objections to ending the phase. When the phase's objection count goes to zero, it indicates to the phasing mechanism that every participant agrees the phase should be ended.

The details on objection handling are fairly complex, and they incur high overhead. Generally, it is recommended to only use the built-in objection objects that govern UVM end-of-phase. It is not recommended to create and use your own objections.

Note: Objection count propagation is limited to components and sequences. Other object types may participate, but they must use a component or sequence object as context.

## Interfaces

The uvm\_objection class has three interfaces or APIs.

### Objection Control

Methods for raising and dropping objections and for setting the drain time.

- `raise_objection ( uvm_object obj = null, string description = "", int count = 1).`

Raises the number of objections for the source object by count, which defaults to 1. The raise of the objection is propagated up the hierarchy.

- `drop_objection ( uvm_object obj = null, string description = "", int count = 1).`

Drops the number of objections for source object by count, which defaults to 1. The drop of the objection is propagated up the hierarchy. If the objection count drops to 0 at any component, an optional `drain_time` and that component's `all_dropped()` callback is executed first. If the objection count is still 0 after this, propagation proceeds to the next level up the hierarchy.

- `set_drain_time ( uvm_object obj = null, time drain).`

Sets the drain time on the given object.

### Recommendations:

- Use `phase.raise_objection` / `phase.drop_objection` inside a component's phase methods to have that component participate in governing end-of-phase.
- Always provide a description - it helps with debug
- Usually use the default count value.
- Limit use of `drain_time` to uvm\_top or the top-level test, if used at all.

## Objection Status

Methods for obtaining status information regarding an objection.

- `get_objection_count ( uvm_object obj)`  
Returns objection count explicitly raised by the given object.
- `get_objection_total ( uvm_object obj = null)`  
Returns objection count for object and all children.
- `get_drain_time ( uvm_object obj)`  
Returns drain time for object ( default: 0ns).
- `display_objections ( uvm_object obj = null, bit show_header = 1)`  
Displays objection information about object.

### Recommendations:

- Generally only useful for debug
- Add `+UVM_OBJECTION_TRACE` to the `vsim` command line to turn on detailed run-time objection tracing. This enables debug without having to modify code and recompile.
- Also add `+UVM_PHASE_TRACE` to augment objection tracing when debugging phase transition issues.

## Callback Hooks

The following callbacks are defined for all `uvm_component`-based objects.

- `raised()`  
Called upon each `raise_objection` by this component or any of its children.
- `dropped()`  
Called upon each `raise_objection` by this component or any of its children.
- `all_dropped()`  
Called when `drop_objection` has reached object and the total count for object goes to zero

### Recommendations:

- Do not use callback hooks. They serve no useful purpose, are called repeatedly throughout the simulation degrading simulation performance.

## Objection Mechanics

Objection counts are propagated up the component hierarchy and upon every explicit raise and drop by any component. Two counter values are maintained for each component: a count of its own explicitly raised objections and a count for all objections raised by it and all its children, if any. Thus, a raise by component `mytest` governing the `main_phase` results in an objection count of 1 for `mytest`, and a total (implicit) objection count of 1 for `mytest` and 1 for `uvm_top`, the implicit top-level for all UVM components. If `mytest.myenv.myagent.mysequencer` were to then raise an objection, that results in an objection count of 1 for `mysequencer`, and a total (implicit) objection count of 1 for `mysequencer`, 1 for `myagent`, 1 for `myenv`, 2 for `mytest`, and 2 for `uvm_top`. Dropping objections propagates in the same fashion, except that when the implicit objection count at any level of the component hierarchy reaches 0, propagation up the hierarchy does not proceed until after a user-defined `drain_time` (default: 0) has elapsed and the `all_dropped()` callback for that component has executed. If during this time an objection is re-raised at or below this level of hierarchy, the all-dropped condition is

negated and further hierarchical propagation of the all\_dropped condition aborted.

### **Raising an objection causes the following:**

1. The component or sequence's source (explicit) objection count is increased by the count argument
2. The component or sequence's total (implicit) objection count is increased by the count argument
3. If a component, its raised() callback is called.
4. If parent is non-null, repeat steps 2-4 for parent.

A sequence's parent is the sequencer component that it currently is running on. Propagation does not occur up the sequence hierarchy.

Virtual sequences (whose m\_sequencer handle is null) do not propagate.

### **Dropping an objection causes the following:**

1. The component or sequence's source (explicit) objection count is decreased by the count argument
2. The component or sequence's total (implicit) objection count is decreased by the count argument
3. If a component, its dropped() callback is called.
4. If the total objection count for the object is not zero and parent is non-null, repeat steps 2-4 for parent.
5. If the total objection count for the object is zero, the following is forked (drop\_objection is non-blocking)
  - Wait for drain time to elapse
  - Call all\_dropped() virtual task callback and wait for completion
  - Adjust count argument by any raises or drops that have occurred since. If drop count still non-zero, go to 4

---

# Sequences

---

## Sequences

---

Learn all about Sequences used to encapsulate stimulus, Sequencer/Driver hookup, pipelined protocols

---

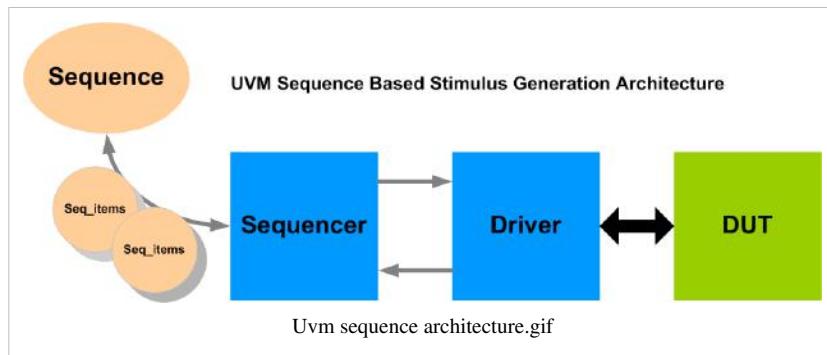
### Sequences Chapter contents:

- Sequences (this page)
- Sequences/Items
- Transaction/Methods
- Sequences/API
- Config/ConfiguringSequences
- Connect/Sequencer
- Driver/Sequence API
- Sequences/Generation
- Sequences/Overrides
- Sequences/Virtual
  - Sequences/VirtualSequencer
- Sequences/Hierarchy
- Sequences/SequenceLibrary
- Driver/Use Models
  - Driver/Unidirectional
  - Driver/Bidirectional
  - Driver/Pipelined
- Sequences/Arbitration
- Sequences/Priority
- Sequences/LockGrab
- Sequences/Slave
- Stimulus/Signal Wait
- Stimulus/Interrupts
- Sequences/Stopping
- Sequences/Layering

## Topic Overview

### Sequence Overview

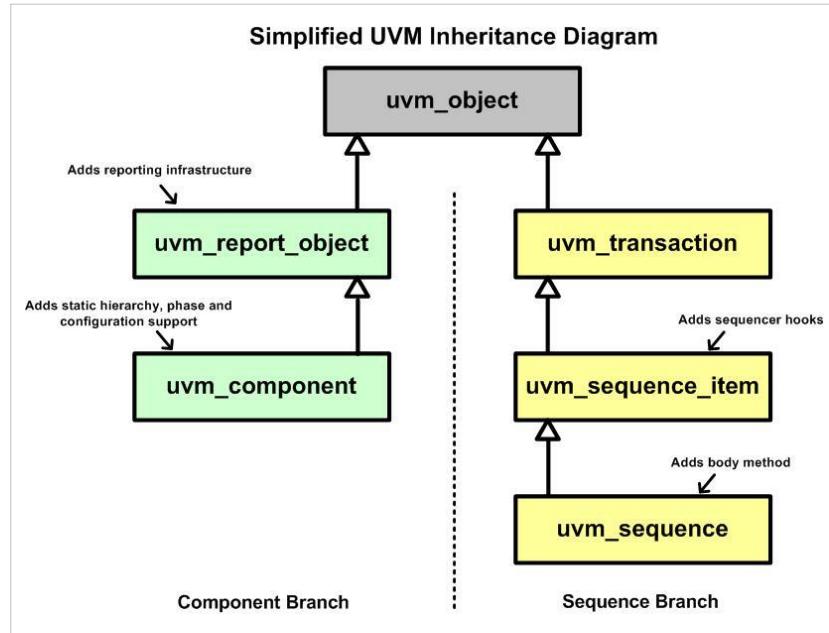
In testbenches written in traditional HDLs like Verilog and VHDL, stimulus is generated by layers of sub-routine calls which either execute time consuming methods (i.e. Verilog tasks or VHDL processes or procedures) or call non-time consuming methods (i.e. functions) to manipulate or generate data. Test cases implemented in these HDLs rely on being able to call sub-routines which exist as elaborated code at the beginning of the simulation. There are several disadvantages with this approach - it is hard to support constrained random stimulus; test cases tend to be 'hard-wired' and quite difficult to change and running a new test case usually requires recompiling the testbench. Sequences bring an Object Orientated approach to stimulus generation that is very flexible and provides new options for generating stimulus.



A sequence is an example of what software engineers call a 'functor', in other words it is an object that is used as a method. An UVM sequence contains a task called body. When a sequence is used, it is created, then the body method is executed, and then the sequence can be discarded. Unlike an `uvm_component`, a sequence has a limited simulation life-time and can therefore be described as a transient object. The sequence body method can be used to create and execute other sequences, or it can be used to generate `sequence_item` objects which are sent to a driver component, via a sequencer component, for conversion into pin-level activity or it can be used to do some combination of the two. The `sequence_item` objects are also transient objects, and they contain the information that a driver needs in order to carry out a pin level interaction with a DUT. When a response is generated by the DUT, then a `sequence_item` is used by the driver to pass the response information back to the originating sequence, again via the sequencer. Creating and executing other sequences is effectively the same as being able to call conventional sub-routines, so complex functions can be built up by chaining together simple sequences.

In terms of class inheritance, the `uvm_sequence` inherits from the `uvm_sequence_item` which inherits from the `uvm_object`. Both base classes are known as objects rather than components. The UVM testbench component hierarchy is built from `uvm_components` which have different properties, which are mainly to do with them being tied into a static component hierarchy as they are built and that component hierarchy stays in place for the life-time of the simulation.

Sequences are the primary means of generating stimulus in the UVM. The fact that sequences and sequence\_items are objects means that they can be easily randomized to generate interesting stimulus. Their object orientated nature also means that they can be manipulated in the same way as any other object. The UVM architecture also keeps these classes separate from the testbench component hierarchy, the advantage being that it is easy to define a new test case by calling and executing different combinations of sequences from a library package without being locked to the methods available within the scope of the components. The disadvantage is that sequences are not able to directly access testbench resources such as configuration information, or handles to register models, which are available in the component hierarchy. Sequences access testbench resources using a sequencer as the key into the component hierarchy.



In the UVM sequence architecture, sequences are responsible for the stimulus generation flow and send sequence\_items to a driver via a sequencer component. The driver is responsible for converting the information contained within sequence\_items into pin level activity. The sequencer is an intermediate component which implements communication channels and arbitration mechanisms to facilitate interactions between sequences and drivers. The flow of data objects is bidirectional, request items will typically be routed from the sequence to the driver and response items will be returned to the sequence from the driver. The sequencer end of the communication interface is connected to the driver end together during the connect phase.

## Sequence Items

As sequence\_items are the foundation on which sequences are built, some care needs to be taken with their design. Sequence\_item content is determined by the information that the driver needs in order to execute a pin level transaction; ease of generation of new data object content, usually by supporting constrained random generation; and other factors such analysis hooks. By convention sequence\_items should also contain a number of standard method implementations to support the use of the object in common transaction operations, these include copy, compare and convert2string.

## Controlling Sequence Execution

Sequence execution is started by calling the `uvm_sequence start()` method, this associates the sequence with a sequencer and then calls the sequences body method. Inside the sequence body method, other sequences can be executed or sequence\_items can be generated and sent to the driver. Generally, stimulus generation is started with one main controlling sequence in the test class `run()` method, and this sequence then spawns other sequences which generate the stimulus within the testbench. Once a chain of sequences is started, the flow of execution can be hierarchical, or parallel, or executed in a randomized order. Sequences can also be randomized to change control or data variables. A

library of sequences can also be created, but it is not recommended.

In order to handle the sequence\_items arriving from the sequence, the driver has access to methods which are implemented in the sequencer, these give the driver several alternate means to indicate that a sequence\_item has been consumed or to send responses back to the sequence.

The handling of sequence\_items inside a sequence often relies to some extent on the how the driver processes sequence items. There are a number of common sequence-driver use models, which include:

- Unidirectional non-pipelined
- Bidirectional non-pipelined
- Pipelined

Warning:

Once a sequence has started execution it should be allowed to complete, if it is stopped prematurely, then there is a reasonable chance that the sequence-sequencer-driver communication mechanism will lock up.

## Sequence Stimulus Generation Variations

### Controlling Stimulus Generation On More Than One Driver

The sequence architecture of a driver, sequencer, sequence\_item and sequence is orientated around a single interface. In most cases, a testbench will need to handle transactions on multiple interfaces and this is achieved through the use of the virtual sequence. A virtual sequence is a sequence that can start sub-sequences on multiple sequencers. The recommended way to implement a virtual sequence is for it to have handles for the target sequencers on which the sub-sequences run. The legacy approach is to use a virtual sequencer which is a normal sequencer which contains handles for the target sequencers.

### Controlling Multiple Sequences Connected To A Single Driver

Multiple sequences can interact concurrently with a driver. The sequencer has an arbitration mechanism to ensure that only one sequence\_item has access to the driver at any point in time. The choice of which sequence item is sent is dependent on an user selectable sequencer arbitration algorithm. There are five built in sequencer arbitration mechanisms and a hook for a user defined algorithm. A sequence can be started with a priority, this enables some of the arbitration mechanisms to give the correct order of precedence to multiple sequences running on a sequencer.

If responses are being returned from the driver to one of several sequences, the sequence id field in the sequence\_item is used by the sequencer to route the response back to the right sequencer. The response handling code in the driver should use the set\_id\_info call to ensure that any response items have the same sequence id as their originating request.

In some cases, such as processing an interrupt, a sequence needs to gain exclusive access to the driver, in this case a sequence can use the grab or lock methods.

## Layering

In many cases, sequences are used to generate streams of data objects which can be abstracted as layers, serial communication channels being one example and accessing a bus through a register indirection is another. The layering mechanism allows sequences to be layered on top of each other, so that high level layer sequences are translated into lower level sequences transparently. This form of sequence generation allows complex stimulus to be built very rapidly.

## Waiting For A Hardware Event

Whilst the driver takes care of normal hardware synchronisation, a sequence execution flow may need to be synchronised to a hardware event such as a transition on a sideband signal, or an end of reset condition. Rather than extend the driver and add another field to the sequence\_item, it is possible to implement a `wait_for_hardware_event` method in a configuration object that contains a pointer to a virtual interface.

## Interrupt Driven Stimulus

One variation of the waiting for a hardware event theme is to use interrupts to trigger the execution of sequences. This might result in the exclusive execution of an interrupt service routine, or it might emulate the control state machine for a hardware device that generates interrupts when it is ready or has completed a task.

# Sequences/Items

The UVM stimulus generation process is based on sequences controlling the behaviour of drivers by generating sequence\_items and sending them to the driver via a sequencer. The framework of the stimulus generation flow is built around the sequence structure for control, but the generation data flow uses sequence\_items as the data objects.

## Data Property Members

The content of the sequence\_item is closely related to the needs of the driver. The driver relies on the content of the sequence\_items it receives to determine which type of pin level transaction to execute. The sequence items property members will consist of data fields that represent the following types of information:

- Control - i.e. What type of transfer, what size
- Payload - i.e. The main data content of the transfer
- Configuration - i.e. Setting up a new mode of operation, error behaviour etc
- Analysis - i.e. Convenience fields which aid analysis - time stamps, rolling checksums etc

## Randomization Considerations

Sequence\_items are randomized within sequences to generate traffic data objects. Therefore, stimulus data properties should generally be declared as rand, and the sequence\_item should contain any constraints required to ensure that the values generated by default are legal, or are within sensible bounds. In a sequence, sequence\_items are often randomized using in-line constraints which extend these base level constraints.

As sequence\_items are used for both request and response traffic and a good convention to follow is that request properties should be rand, and that response properties should not be rand. This optimises the randomization process and also ensures that any collected response information is not corrupted by any randomization that might take place.

For example consider the following bus protocol sequence\_item:

```
class bus_seq_item extends uvm_sequence_item;

  // Request data properties are rand
  rand logic[31:0] addr;
  rand logic[31:0] write_data;
  rand bit read_not_write;
  rand int delay;

  // Response data properties are NOT rand
  bit error;
  logic[31:0] read_data;

  `uvm_object_utils(bus_seq_item)

  function new(string name = "bus_seq_item");
    super.new(name);
  endfunction
```

```
// Delay between bus cycles is in a sensible range
constraint at_least_1 { delay inside {[1:20]};}

// 32 bit aligned transfers
constraint align_32 {addr[1:0] == 0;}

// etc
endclass: bus_seq_item
```

## Sequence Item Methods

The uvm\_sequence\_item inherits from the uvm\_object via the uvm\_transaction class. The uvm\_object has a number of virtual methods which are used to implement common data object functions (copy, clone, compare, print, transaction recording) and these should be **implemented** to make the sequence\_item more general purpose.

A sequence\_item is often used in analysis traffic and it may be useful to add utility functions which aid functional coverage or analysis.

# Transaction/Methods

When working with data object classes derived from uvm\_objects, including ones derived from uvm\_transactions, uvm\_sequence\_items and uvm\_sequences, there are a number of methods which are defined for common operations on the data objects properties. In turn, each of these methods calls one or more virtual methods which are left for the user to implement according to the detail of the data members within the object. These methods and their corresponding virtual methods are summarised in the following table.

Method called by user	Virtual method	Purpose
copy	do_copy	Performs a deep copy of an object
clone	do_copy	Creates a new object and then does a deep copy of an object
compare	do_compare	Compares one object to another of the same type
convert2string	-	Returns a string representation of the object
print	do_print	Prints the result of convert2string to the screen
sprint	do_print	Returns the result of convert2string
record	do_record	Handles transaction recording
pack	do_pack	Compresses object contents into a bit format
unpack	do_unpack	Converts a bit format into the data object format

The do\_xxx methods can be implemented and populated using `uvm\_field\_xxx macros, but the resultant code is inefficient, hard to debug and can be prone to error. The recommended approach is to implement the methods manually which will result in improvements in testbench performance and memory footprint. For more information on the issues involved see the page on **macro cost benefit analysis**.

Consider the following sequence\_item which has properties in it that represent most of the common data types:

```
class bus_item extends uvm_sequence_item;

// Factory registration
`uvm_object_utils(bus_item)

// Properties - a selection of common types:
rand int delay;
rand logic[31:0] addr;
rand op_code_enum op_code;
string slave_name;
rand logic[31:0] data[];
bit response;

function new(string name = "bus_item");
    super.new(name);
endfunction

endclass: bus_item
```

The common methods that need to be populated for this sequence\_item are:

### do\_copy

The purpose of the do\_copy method is to provide a means of making a deep copy\* of a data object. The do\_copy method is either used on its own or via the uvm\_objects clone() method which allows independent duplicates of a data object to be created. For the sequence\_item example, the method would be implemented as follows:

```
// do_copy method:
function void do_copy(uvm_object rhs);

bus_item rhs_;

if(!$cast(rhs_, rhs)) begin
  uvm_report_error("do_copy:", "Cast failed");
  return;
end

super.do_copy(rhs); // Chain the copy with parent classes
delay = rhs_.delay;
addr = rhs_.addr;
op_code = rhs_.op_code;
slave_name = rhs_.slave_name;
data = rhs_.data;
response = rhs_.response;
endfunction: do_copy

// Example of how do_copy would be used:

// Directly:
bus_item A, B;

A.copy(B); // A becomes a deep copy of B

// Indirectly:
$cast(A, B.clone()); // Clone returns an uvm_object which needs
// to be cast to the actual type
```

Note that the rhs argument is of type uvm\_object since it is a virtual method, and that it therefore needs to be cast to the actual transaction type before its fields can be copied. A design consideration for this method is that it may not always make sense to copy all property values from one object to another.

*\*A deep copy is one where the value of each of the individual properties in a data object are copied to another, as opposed to a shallow copy where just the data pointer is copied.*

## do\_compare

The do\_compare method is called by the uvm\_object compare() method and it is used to compare two data objects of the same type to determine whether their contents are equal. The do\_compare() method should only be coded for those properties which can be compared.

The uvm\_comparer policy object has to be passed to the do\_compare() method for compatibility with the virtual method template, but it is not necessary to use it in the comparison function and performance can be improved by not using it.

```
// do_compare implementation:
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  bus_item rhs_;

  // If the cast fails, comparison has also failed
  // A check for null is not needed because that is done in the compare()
  // function which calls do_compare()
  if(!$cast(rhs_, rhs)) begin
    return 0;
  end

  return((super.do_compare(rhs, comparer) &&
    (delay == rhs_.delay) &&
    (addr == rhs_.addr) &&
    (op_code == rhs_.op_code) &&
    (slave_name == rhs_.slave_name) &&
    (data == rhs_.data) &&
    (response == rhs_.response));
endfunction: do_compare

// Useage example - do_compare is not used directly
bus_item A, B;

if(!A.compare(B)) begin
  // Report and handle error
end
else begin
  // Report and handle success
end
```

## convert2string

In order to get debug or status information printed to a simulator transcript or a log file from a data object there needs to be a way to convert the objects contents into a string representation - this is the purpose of the convert2string() method. Calling the method will return a string detailing the values of each of the properties formatted for transcript display or for writing to a file. The format is determined by the user:

```
// Implementation example:
function string convert2string();
  string s;

  s = super.convert2string();
  // Note the use of \t (tab) and \n (newline) to format the data in columns
  // The enumerated op_code types .name() method returns a string corresponding to its value
  $sformat(s, "%s\n delay \t%0d\n addr \t%0h\n op_code \t%s\n slave_name \t%s\n",
    s, delay, addr, op_code.name(), slave_name);
  // For an array we need to iterate through the values:
  foreach(data[i]) begin
    $sformat(s, "%s data[%0d] \t%0h\n", s, i, data[i]);
  end
  $sformat(s, "%s response \t%0b\n", s, response);
  return s;
endfunction: convert2string
```

## do\_print

The do\_print() method is called by the uvm\_object print() method. Its original purpose was to print out a string representation of an uvm data object using one of the uvm\_printer policy classes. However, a higher performance version of the same functionality can be achieved by implementing the method as a wrapper around a \$display() call that takes the string returned by a convert2string() call as an argument:

```
function void do_print(uvm_printer printer);
  if(printer.knobs.sprint == 0) begin
    $display(convert2string());
  end
  else begin
    printer.m_string = convert2string();
  end
endfunction: do_print
```

This implementation avoids the use of the uvm\_printer policy classes, takes less memory and gives higher performance. However, this is at the expense of not being able to use the various formatted uvm printer options.

***To achieve full optimisation, avoid using the print() and sprint() methods all together and call the convert2string() method directly.***

## do\_record

The do\_record() method is intended to support the viewing of data objects as transactions in a waveform GUI. Like the printing data object methods, the principle is that the fields that are recorded are visible in the transaction viewer. The underlying implementation of the `uvm\_record\_field macro used in the do\_record() method is simulator specific and for Questa involves the use of the \$add\_attribute() system call:

```
function void do_record(uvm_recorder recorder);
    super.do_record(recorder); // To record any inherited data members
    `uvm_record_field("delay", delay)
    `uvm_record_field("addr", addr)
    `uvm_record_field("op_code", op_code.name())
    `uvm_record_field("slave_name", slave_name)
    foreach(data[i]) begin
        `uvm_record_field($sformatf("data[%0d]", i), data[i])
    end
    `uvm_record_field("response", response)
endfunction: do_record
```

In order to get transaction viewing to work with Questa you need to:

- Implement the do\_record() method as shown
- Set the recording\_detail config item to UVM\_FULL:

```
set_config_int("*", "recording_detail", UVM_FULL);
```

The transactions that are recorded by implementing do\_record() and by turning on the recording\_detail are available in the sequencer with a transaction stream handle name of aggregate\_items.

## do\_pack and do\_unpack

These two methods are not commonly used and their purpose is to convert a data object into a bit stream (i.e. an integer) to allow a representation to be passed between language domains, for instance between SystemVerilog and C/C++. The recommended implementation of these two methods is likely to change with forthcoming versions of Questa and are not documented here. However, they are documented in the paper which can be found on the page ***macro cost benefit analysis***.

# Sequences/API

---

## Sequence API Fundamentals

A uvm\_sequence is derived from an uvm\_sequence\_item and it is parameterised with the type of sequence\_item that it will send to a driver via a sequencer. The two most important properties of a sequence are the body method and the m\_sequencer handle.

### The body Method

An uvm\_sequence contains a task method called body. It is the content of the body method that determines what the sequence does.

### The m\_sequencer Handle

When a sequence is started it is associated with a sequencer. The m\_sequencer handle contains the reference to the sequencer on which the sequence is running. The m\_sequencer handle can be used to access configuration information and other *resources* in the UVM component hierarchy.

## Running a sequence

To get a sequence to run there are three steps that need to occur:

### Step 1 - Sequence Creation

Since the sequence is derived from an uvm\_object, it is created using the factory:

```
my_sequence m_seq; // Declaration

m_seq = my_sequence::type_id::create("m_seq");
```

Using the factory creation method allows the sequence to be overridden with a sequence of derived type as a means of varying the stimulus generation.

### Step 2 - Sequence Configuration

The sequence may need to be configured - examples of configuration include:

- Setting up start values - e.g. a start address or data value
- Setting up generation loop variables - e.g. number of iterations, which index number to start from
- Setting up pointers to testbench resources - e.g. a register map

```
m_seq.no_iterations = 10; // Direct assignment of values

// Using randomization
if(!m_seq.randomize() with {no_iterations inside {[5:20]}}) begin
  `uvm_error("marker", "Randomization failure for m_seq")
end

// Assigning a test bench resource
m_seq.reg_map = env.reg_map;
```

### Step 3 - Starting The Sequence

A sequence is started using a call to its start() method, passing as an argument a pointer to the sequencer through which it will be sending sequence\_items to a driver. The start() method assigns the sequencer pointer to a sequencer handle called m\_sequencer within the sequence and then calls the body task within the sequence. When the sequence body task completes, the start method returns. Since it requires the body task to finish and this requires interaction with a driver, start() is a blocking method.

The start method has three optional arguments which have defaults which means that most of the time they are not necessary:

```
virtual task start (uvm_sequencer_base sequencer, // Pointer to sequencer
                    uvm_sequence_base parent_sequencer = null, // Relevant if called within a sequence
                    integer this_priority = 100, // Priority on the sequencer
                    bit call_pre_post = 1); // pre_body and post_body methods called

// For instance - called from an uvm_component - usually the test:
apb_write_seq.start(env.m_apb_agent.m_sequencer);

// Or called from within a sequence:
apb_compare_seq.start(m_sequencer, this);
```

It is possible to call the sequence start method without any arguments, and this will result in the sequence running without a direct means of being able to connect to a driver.

### Sending a sequence\_item to a driver

To send a sequence\_item to a driver there are four steps that need to occur:

#### Step 1 - Creation

The sequence\_item is derived from uvm\_object and should be created via the factory:

Using the factory creation method allows the sequence\_item to be overridden with a sequence\_item of a derived type if required.

#### Step 2 - Ready - start\_item()

The start\_item() call is made, passing the sequence\_item handle as an argument. This call blocks until the sequencer grants the sequence and the sequence\_item access to the driver.

### Step 3 - Set

The sequence\_item is prepared for use, usually through randomization, but it may also be initialised by setting properties directly.

### Step 4 - Go - finish\_item()

The finish\_item() call is made, which blocks until the driver has completed its side of the transfer protocol for the item. No simulation time should be consumed between start\_item() and finish\_item().

### Step 5 - Response - get\_response()

This step is optional, and is only used if the driver sends a response to indicate that it has completed transaction associated with the sequence\_item. The get\_response() call blocks until a response item is available from the sequencer's response FIFO.

```
// Inside some sequence connected to a sequencer

my_sequence_item item;

task body;
    // Step 1 - Creation
    item = my_sequence_item::type_id::create("item");

    // Step 2 - Ready - start_item()
    start_item(item);
    // Step 3 - Set
    if(!item.randomize() with {address inside {[0:32'h4FFF_FFFF]};}) begin
        `uvm_error("body", "randomization failure for item")
    end
    // Step 4 - Go - finish_item()
    finish_item(item);
endtask: body
```

### Late Randomization

In the sequence\_item flow above, steps 2 and 3 could be done in any order. However, leaving the randomization of the sequence\_item until just before the finish\_item() method call has the potential to allow the sequence\_item to be randomized according to conditions true at the time of generation. This is sometimes referred to as late randomization. The alternative approach is to generate the sequence\_item before the start\_item() call, in this case the item is generated before it is necessarily clear how it is going to be used.

In previous generation verification methodologies, such as Specman and the AVM, generation was done at the beginning of the simulation and a stream of pre-prepared sequence\_items was sent across to the driver. With late randomization, sequence\_items are generated just in time and on demand.

## Coding Guidelines

### Make sequence input variables rand, output variables non-rand

A good coding convention to follow is to make variables which are used as inputs by sequence random and to leave variables which are to collect output or response information as non-random. This ensures that a sequence can be configured by randomization with constraints. The same convention applies to *sequence\_items*.

Obvious exceptions to this convention would be handles and strings within the sequence.

### Do not use pre/\_post\_body in sequences

The pre\_body() method is intended to be executed before the body method and the post\_body() method is intended to be called after body has completed. In practice, the functionality that these two methods might contain can easily be put directly into the body method. If there is some common initialisation work to be done, then this can be put into the body method of a base class and called using super.body().

The way in which sequences are started affects whether these methods are called or not. Using start\_item(), finish\_item() means that the methods are not called, and using start() can mean that they are not called. It is therefore easier for the user if the pre/\_post\_body methods are not used.

### Use start for sequences, start\_item, finish\_item for sequence\_items

Sequence\_items should be sent to the driver using the start\_item() and finish\_item() calls. Sequences should be started using the start() call.

#### Justification:

Although sequences can also be started using start\_item() and finish\_item(), it is clearer to the user whether a sequence or a sequence item is being processed if this convention is followed.

Using the start() call also allows the user to control whether the sequences pre\_body() and post\_body() hook methods are called. By default, the start() method enables the calling of the pre and post\_body() methods, but this can be disabled by passing an extra argument into the start() method. Also note that the start\_item() and finish\_item() calls do not call pre or post\_body().

Using start() for sequences means that a user does not need to know whether a sub-sequence contains one of these hook methods.

### Sequences should not directly consume time

Sequence code should not explicitly consume time by using delay statements. They should only consume time by virtue of the process of sending sequence\_items to a driver.

#### Justification:

Keeping this separation allows sequences to be reused as stimulus when an UVM testbench is linked to an emulator for hardware acceleration.

# Connect/Sequencer

The transfer of request and response sequence items between sequences and their target driver is facilitated by a bidirectional TLM communication mechanism implemented in the sequencer. The `uvm_driver` class contains an `uvm_seq_item_pull_port` which should be connected to an `uvm_seq_item_pull_export` in the sequencer associated with the driver. The port and export classes are parameterised with the types of the `sequence_items` that are going to be used for request and response

transactions. Once the port-export connection is made, the driver code can use the API implemented in the export to get request `sequence_items` from sequences and return responses to them.

The connection between the driver port and the sequencer export is made using a TLM connect method during the connect phase:

```
// Driver parameterised with the same sequence_item for request & response
// response defaults to request
class adpcm_driver extends uvm_driver #(adpcm_seq_item);
...
endclass: adpcm_driver

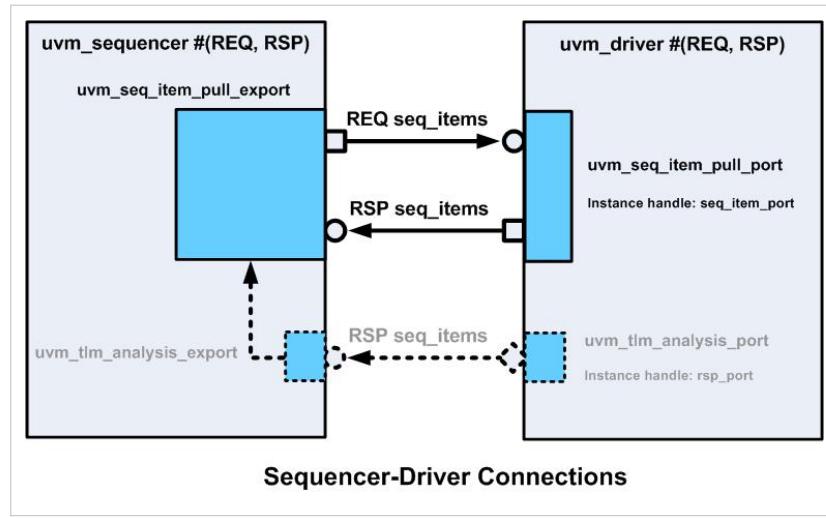
// Sequencer parameterised with the same sequence item for request & response
class adpcm_sequencer extends uvm_sequencer #(adpcm_seq_item);
...
endclass: adpcm_sequencer

// Agent containing a driver and a sequencer - uninteresting bits left out
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
adpcm_sequencer m_sequencer;
adpcm_agent_config m_cfg;

// Sequencer-Driver connection:
function void connect_phase(uvm_phase phase);
  if(m_cfg.active == UVM_ACTIVE) begin // The agent is actively driving stimulus
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export); // TLM connection
    m_driver.vif = cfg.vif; // Virtual interface assignment
  end
endfunction: connect_phase

```



```
    end
endfunction: connect_phase
```

The connection between a driver and a sequencer is typically made in the `connect_phase()` method of an agent. With the standard UVM driver and sequencer base classes, the TLM connection between a driver and sequencer is a one to one connection - multiple drivers are not connected to a sequencer, nor are multiple sequencers connected to a driver.

In addition to this bidirectional TLM port, there is an `analysis_port` in the driver which can be connected to an `analysis_export` in the sequencer to implement a unidirectional response communication path between the driver and the sequencer. This is a historical artifact and provides redundant functionality which is not generally used. The bidirectional TLM interface provides all the functionality required. If this analysis port is used, then the way to connect it is as follows:

```
// Same agent as in the previous bidirectional example:
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
adpcm_sequencer m_sequencer;
adpcm_agent_config m_cfg;

// Connect method:
function void connect_phase(uvm_phase phase );
  if(m_cfg.active == UVM_ACTIVE) begin
    m_driver.seq_item_port.connect(m_sequencer.seq_item_export); // Always need this
    m_driver.rsp_port.connect(m_sequencer.rsp_export); // Response analysis port connection
    m_driver.vif = cfg.vif;
  end
  //...
endfunction: connect_phase

endclass: adpcm_agent
```

Note that the bidirectional TLM connection will always have to be made to effect the communication of requests.

One possible use model for the `rsp_port` is to notify other components when a driver returns a response, otherwise it is not needed.

# Driver/Sequence API

---

The uvm\_driver is an extension of the uvm\_component class that adds an uvm\_seq\_item\_pull\_port which is used to communicate with a sequence via a sequencer. The uvm\_driver is a parameterised class and it is parameterised with the type of the request sequence\_item and the type of the response sequence\_item. In turn, these parameters are used to parameterise the uvm\_seq\_item\_pull\_port. The fact that the response sequence\_item can be parameterised independently means that a driver can return a different response item type from the request type. In practice, most drivers use the same sequence item for both request and response, so in the source code the response sequence\_item type defaults to the request sequence\_item type.

The use model for the uvm\_driver class is that it consumes request (REQ) sequence\_items from the sequencers request FIFO using a handshaked communication mechanism, and optionally returns response (RSP) sequence\_items to the sequencers response FIFO. The handle for the seq\_item\_pull\_port within the uvm\_driver is the seq\_item\_port. The API used by driver code to interact with the sequencer is referenced by the seq\_item\_port, but is actually implemented in the sequencers seq\_item\_export (this is standard TLM practice).

## UVM Driver API

The driver sequencer API calls are:

### **get\_next\_item**

This method blocks until a REQ sequence\_item is available in the sequencers request FIFO and then returns with a pointer to the REQ object.

The get\_next\_item() call implements half of the driver-sequencer protocol handshake, and it must be followed by an item\_done() call which completes the handshake. Making another get\_next\_item() call before issuing an item\_done() call will result in a protocol error and driver-sequencer deadlock.

### **try\_next\_item**

This is a non-blocking variant of the get\_next\_item() method. It will return a null pointer if there is no REQ sequence\_item available in the sequencers request FIFO. However, if there is a REQ sequence\_item available it will complete the first half of the driver-sequencer handshake and must be followed by an item\_done() call to complete the handshake.

### **item\_done**

The non-blocking item\_done() method completes the driver-sequencer handshake and it should be called after a get\_next\_item() or a successful try\_next\_item() call.

If it is passed no argument or a null pointer it will complete the handshake without placing anything in the sequencer's response FIFO. If it is passed a pointer to a RSP sequence\_item as an argument, then that pointer will be placed in the sequencer's response FIFO.

## peek

If no REQ sequence\_item is available in the sequencer's request FIFO, the peek() method will block until one is available and then return with a pointer to the REQ object, having executed the first half of the driver-sequencer handshake. Any further calls to peek() before a get() or an item\_done() call will result in a pointer to the same REQ sequence item being returned.

## get

The get() method blocks until a REQ sequence\_item is available in the sequencer's request FIFO. Once one is available, it does a complete protocol handshake and returns with a pointer to the REQ object.

## put

The put() method is non-blocking and is used to place a RSP sequence\_item in the sequencer's response FIFO.

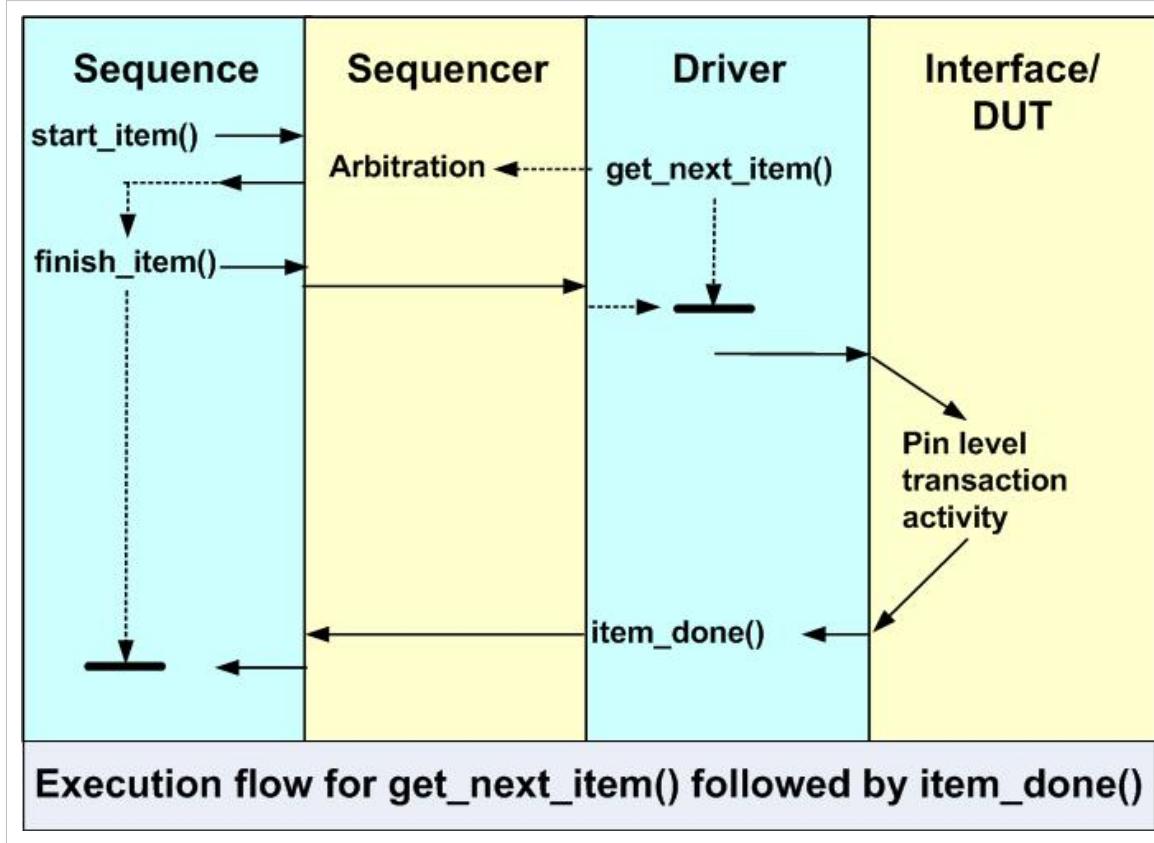
The put() method can be called at any time and is not connected with the driver-sequencer request handshaking mechanism.

**Note:** The get\_next\_item(), get() and peek() methods initiate the *sequencer arbitration* process, which results in a sequence\_item being returned from the active sequence which has selected. This means that the driver is effectively pulling sequence\_items from the active sequences as it needs them.

## Recommended Driver-Sequencer API Use Models

The purpose of the driver sequencer API is for the driver to receive a series of sequence\_items from sequences containing the data required to initiate transactions, and for the driver to communicate back to the sequence that it has finished with the sequence\_item and that it is ready for the next item. There are two common ways of doing this:

### get\_next\_item() followed by item\_done()



This use model allows the driver to get a sequence item from a sequence, process it and then pass a hand-shake back to the sequence using `item_done()`. No arguments should be passed in the `item_done()` call. This is the preferred driver-sequencer API use model, since it provides a clean separation between the driver and the sequence.

```
//
// Driver run method
//
task run_phase( uvm_phase phase );
    bus_seq_item req_item;

    forever begin
        seq_item_port.get_next_item(req_item); // Blocking call returning the next transaction
        @(posedge vif.clk);
        vif.addr = req_item.address; // vif is the drivers Virtual Interface
    //
    // etc

```

```

// 
// End of bus cycle
if(req_item.read_or_write == READ) begin // Assign response data to the req_item fields
    req_item.rdata = vif.rdata;
end
req_item.resp = vif.error; // Assign response to the req_item response field
seq_item_port.item_done(); // Signal to the sequence that the driver has finished with the item
end
endtask: run

```

The corresponding sequence implementation would be a `start_item()` followed by a `finish_item()`. Since both the driver and the sequence are pointing to the same `sequence_item`, any data returning from the driver can be referenced within the sequence via the `sequence_item` handle. In other words, when the handle to a `sequence_item` is passed as an argument to the `finish_item()` method the drivers `get_next_item()` method call completes with a pointer to the same `sequence_item`. When the driver makes any changes to the `sequence_item` it is really updating the object inside the sequence. The drivers call to `item_done()` unblocks the `finish_item()` call in the sequence and then the sequence can access the fields in the `sequence_item`, including those which the driver may have updated as part of the response side of the pin level transaction.

```

// 
// Sequence body method:
// 
task body();
    bus_seq_item req_item;
    bus_seq_item req_item_c;

    req_item = bus_seq_item::type_id::create("req_item");

    repeat(10) begin
        $cast(req_item_c, req_item.clone); // Good practice to clone the req_item item
        start_item(req_item_c);
        req_item_c.randomize();
        finish_item(req_item_c); // Driver has returned REQ with the response fields updated
        `uvm_report("body", req_item_c.convert2string());
    end
endtask: body

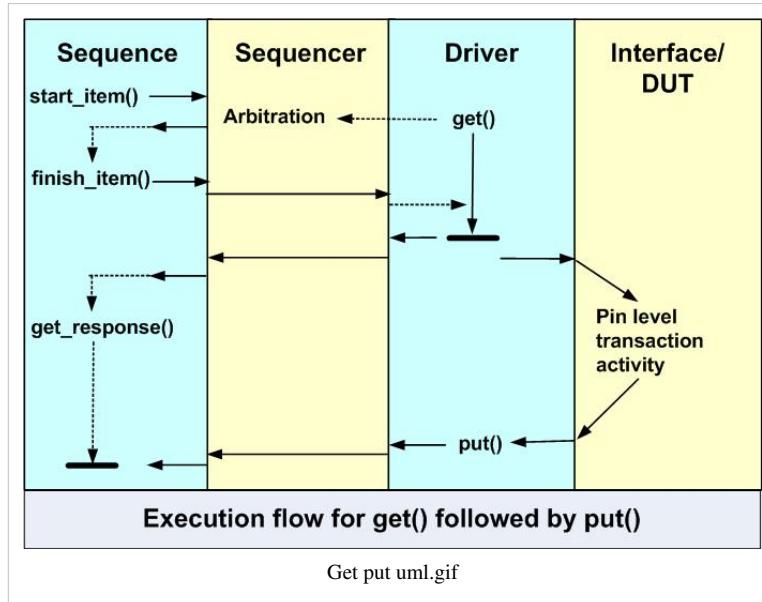
```

## get(req) followed by put(rsp)

With this use model, the driver does a get(req) which does the getting of the next sequence\_item and sends back the sequence handshake in one go, before the driver has had any time to process the sequence\_item. The driver uses the put(rsp) method to indicate that the sequence\_item has been consumed and to return a response. The driver response handling should follow the guidelines given in the next section.

If this use model is followed, the sequence needs to follow the finish\_item() call with a get\_response() call which will block until the driver calls put(rsp).

The drawbacks of this use model are that it is more complicated to implement on the driver side and that the sequence writer always has to remember to handle the response that is returned.



```

// run method within the driver
//
task run_phase( uvm_phase phase );
    REQ req_item; //REQ is parameterized type for requests
    RSP rsp_item; //RSP is parameterized type for responses

    forever begin
        seq_item_port.get(req_item); // finish_item in sequence is unblocked
        @(posedge vif.clk);
        vif.addr = req_item.addr;
        //
        // etc
        //
        // End of bus transfer
        $cast(rsp_item, req_item.clone()); // Create a response transaction by cloning req_item
        rsp_item.set_id_info(req_item); // Set the rsp_item sequence id to match req_item
        if(req_item.read_or_write == READ) begin // Copy the bus data to the response fields

```

```

    rsp_item.read_data = vif.rdata;
  end
  rsp_item.resp = vif.error;
  seq_item_port.put(rsp_item); // Handshake back to the sequence via its get_response() call
end
endtask

// 
// Corresponding code within the sequence body method
//
task body();
  REQ req_item; //REQ is parameterized type for requests
  RSP rsp_item; //RSP is parameterized type for responses

  repeat(10) begin
    req_item = bus_seq_item::type_id::create("req_item");
    start_item(req_item);
    req_item.randomize();
    finish_item(req_item); // This passes to the driver get() call and is returned immediately
    get_response(rsp_item); // Block until a response is received
    `uvm_info("body", rsp_item.convert2string(), UVM_LOW);
  end
endtask: body

```

### Routing response items back to the parent sequence

One of the complications of this use model arises when there are several sequences communicating with a driver through a sequencer. The sequencer looks after which request sequence\_item gets routed to the driver from which sequence. However, when the driver creates a response sequence\_item it needs to be routed back to the right sequence. The way in which the UVM takes care of this problem is that each sequence\_item has a couple of id fields, one for the parent sequence and one to identify the sequence\_item, these fields are used by the sequencer to route responses back to the parent sequence. A request sequence\_item has these fields set by the sequencer as a result of the start\_item() method, therefore, a new response sequence\_item needs to take a copy of the requests id information so that it can be routed back to the originating sequence. The set\_id\_info() method is provided in the uvm\_sequence\_item base class for this purpose.

## Coding Guidelines For Driver Response Items

### set\_id\_info

The uvm\_sequence\_item contains an id field which is set by a sequencer during the sequence start\_item() call. This id allows the sequencer to keep track of which sequence each sequence\_item is associated with, and this information is used to route response items back to the correct sequence. Although in the majority of cases only one sequence is actively communicating with a driver, the mechanism is always in use. The sequence\_item set\_id\_info method is used to set a response item id from a the id field in request item.

If a request sequence\_item is returned then the sequence id is already set. However, when a new or cloned response item is created, it must have its id set.

## Use Clone For Pointer Safety

When a response item is sent back from a driver to a sequence, its pointer will be stored in the sequencer's response FIFO. If a response is not consumed before the next response pointer is sent then, unless the new response pointer is for a new object, both pointers will be referencing the same object. A common symptom of this problem is when successive reads of the FIFO yield objects with the same content.

The way to prevent this occurring is to clone the response item so that a new object is created and a pointer to this new object is passed to the sequencer response FIFO or to have different request and response types.

```
// Somewhere in a driver - request and response items
bus_seq_item req_item;
bus_seq_item rsp_item;

task run_phase( uvm_phase phase );
    forever begin
        seq_item_port.get(req_item);
        assert($cast(rsp_item, req_item.clone())); // This does not copy the id info
        rsp.set_id_info(req_item); // This sets the rsp_item id to the req_item id
        //
        // Do the pin level transaction, populate the response fields
        //
        // Return the response:
        seq_item_port.put(rsp_item);
        //
    end
endtask: run
```

# Sequences/Generation

The uvm\_sequence\_base class extends the uvm\_sequence\_item class by adding a body task method. The sequence is used to generate stimulus through the execution of its body task. A sequence object is designed to be a transient dynamic object which means that it can be created, used and then garbage collected after it has been dereferenced.

The use of sequences in the UVM enables a very flexible approach to stimulus generation. Sequences are used to control the generation and flow of sequence\_items into drivers, but they can also create and execute other sequences, either on the same driver or on a different one. Sequences can also mix the generation of sequence\_items with the execution of sub-sequences. Since sequences are objects, the judicious use of polymorphism enables the creation of interesting randomized stimulus generation scenarios.

In any sequence stimulus generation process there are three primary layers in the flow:

1. **The master control thread** - This may be a run task in an UVM test component or a high level sequence such as a virtual sequence or a default sequence. The purpose of this thread is to start the next level of sequences.
2. **The individual sequences** - These may be stand-alone sequences that simply send sequence\_items to a driver or they may in turn create and execute sub-sequences.
3. **The sequence\_item** - This contains the information that enables a driver to perform a pin level transaction. The sequence item contains rand fields which are randomized with constraints during generation within a sequence.

## Sequence Execution Flow Alternatives

For any given sequence flow there are three basic models which may be mixed together to create different execution graphs.

### Linear Execution Flow

With a linear flow, sequences are executed in order, with each sequence completing before the next one is started.

### Parallel Execution Flow

In a parallel flow, fork-join is used to execute sequences in parallel. This means that two or more sequences may be interacting with a driver at any point in time. The SystemVerilog join\_any and join\_none constructs allow sequence processes to be spawned and left executing to overlap with subsequent sequences in the flow.

#### Coding Guideline - Allow fork-joined sequences to complete

A sequence must complete all its transfers before it is terminated, otherwise the sequencer handshake mechanism will be violated and the sequencer will lock up.

Using fork join\_any/join\_nones with sequences requires some care and attention to ensure that this rule is followed.

#### Do not fork, join\_any, disable fork;

Using a fork <multiple\_sequences> join\_any followed by a disable fork in a sequence will result in the uncompleted sequence threads being terminated which will lock up the sequencer.

```
//  
// DO NOT USE THIS PATTERN - Supplied as an example of what NOT to do  
//
```

```

// Parent sequence body method running child sub_sequences within a fork join_any
task body();
//
// Code creating and randomizing the child sequences
//
fork
    seq_A.start(m_sequencer);
    seq_B.start(m_sequencer);
    seq_C.start(m_sequencer);
join_any
// First past the post completes the fork and disables it
 disable fork;
// Assuming seq_A completes first - seq_B and seq_C will be terminated in an indeterminate
// way, locking up the sequencer

```

### Do not use fork join\_none to exit a body method

Using fork, join\_none around a body method to enable it to be executed in parallel with other sequences results in the sequences start method being terminated before the body method content has started communication with the sequencer. This coding style can be avoided by using fork join\_none in the higher level control thread.

```

//
// DO NOT USE THIS PATTERN - Supplied as an example of what NOT to do
//
// Inside sequence_As body method
//
task body();
// Initialise etc
    fork
        // The body code including other sequences and sequence_items
        // ...
    join_none
endtask body;
//
// The body task exits immediately, in the controlling thread the idea is that sequence_A executes
// in parallel with other sequences:
//
task run_phase( uvm_phase phase );
// ....
    sequence_A_h.start(m_sequencer);
    another_seq_h.start(m_sequencer); // This sequence executes in parallel with sequence_A_h
// ...
//
// The way to achieve the desired functionality is to remove the fork join_none from sequence_A
// and to fork join the two sequences in the control thread:
//

```

```

task run;
// ....
fork
  sequence_A_h.start(m_sequencer);
  another_seq_h.start(m_sequencer);
join
// ....

```

### Do not fork join a sequence which contains a forever loop

If a sequence contains a forever loop within its body method, it will loop indefinitely. If this sequence is started within a fork join of parent sequence and the parent sequence terminates, then the child sequences body method will continue to loop. If the parent sequence is terminated by a disable fork, then the child sequence may be caught mid-handshake and this will lock up the sequencer.

### Hierarchical Execution Flow

A hierarchical flow starts with a top level sequence which creates and executes one or more sub-sequences, which in turn create and execute further sub-sequences. This approach is analogous to using layered software organised top-down so that a high level command is translated into a series of lower level calls until it reaches an atomic level at which bus level commands can be executed.

This layered hierarchical approach is described in the Hierarchy article.

### Exploiting The Sequence As An Object

Sequences are objects which means that they have object characteristics which can be exploited during the stimulus generation process.

### Randomized Fields

Like a sequence\_item, a sequence can contain data fields that can be marked as rand fields. This means that a sequence can be made to behave differently by randomizing its variables before starting it. The use of constraints within the sequence allows the randomization to be within "legal" bounds, and the use of in-line constraints allows either a specific values or values within ranges to be generated.

Typically, the fields that are randomized within a sequence control the way in which it generates stimulus. For instance a sequence that moves data from one block of memory to another would contain a randomized start from address, a start to address and a transfer size. The transfer size could be constrained to be within a system limit - say 1K bytes. When the sequence is randomized, then the start locations would be constrained to be within the bounds of the relevant memory regions.

```

// This sequence shows how data members can be set to rand values
// to allow the sequence to either be randomized or set to a directed
// set of values in the controlling thread
//
// The sequence reads one block of memory (src_addr) into a buffer and then
// writes the buffer into another block of memory (dst_addr). The size

```

```
// of the buffer is determined by the transfer size
//
class mem_trans_seq extends bus_seq_base;

`uvm_object_utils(mem_trans_seq)

// Randomised variables
rand logic[31:0] src_addr;
rand logic[31:0] dst_addr;
rand int transfer_size;

// Internal buffer
logic[31:0] buffer[];

// Legal limit on the page size is 1023 transfers
//
// No point in doing a transfer of 0 transfers
//
constraint page_size {
    transfer_size inside {[1:1024]};
}

// Addresses need to be aligned to 32 bit transfers
constraint address_alignment {
    src_addr[1:0] == 0;
    dst_addr[1:0] == 0;
}

function new(string name = "mem_trans_seq");
    super.new(name);
endfunction

task body;
    bus_seq_item req = bus_seq_item::type_id::create("req");
    logic[31:0] dst_start_addr = dst_addr;

    buffer = new[transfer_size];

    `uvm_info("run:", $sformatf("Transfer block of %0d words from %0h-%0h to %0h-%0h",
        transfer_size, src_addr, src_addr+((transfer_size-1)*4),
        dst_addr, dst_addr+((transfer_size-1)*4)), UVM_LOW)

    // Fill the buffer
    for(int i = 0; i < transfer_size-1; i++) begin
```

```

start_item(req);

if(!req.randomize() with {addr == src_addr; read_not_write == 1; delay < 3;}) begin
  `uvm_error("body", "randomization failed for req")
end

finish_item(req);

buffer[i] = req.read_data;

src_addr = src_addr + 4; // Increment to the next location
end

// Empty the buffer

for(int i = 0; i < transfer_size-1; i++) begin
  start_item(req);

  if(!req.randomize() with {addr == dst_addr; read_not_write == 0; write_data == buffer[i]; delay < 3;}) begin
    `uvm_error("body", "randomization failed for req")
  end

  finish_item(req);

  dst_addr = dst_addr + 4; // Increment to the next location
end

dst_addr = dst_start_addr;

// Check the buffer transfer

for(int i = 0; i < transfer_size-1; i++) begin
  start_item(req);

  if(!req.randomize() with {addr == dst_addr; read_not_write == 1; write_data == buffer[i]; delay < 3;}) begin
    `uvm_error("body", "randomization failed for req")
  end

  finish_item(req);

  if(buffer[i] != req.read_data) begin
    `uvm_error("run:", $sformatf("Error in transfer @%0h : Expected %0h, Actual %0h", dst_addr, buffer[i], req.read_data))
  end

  dst_addr = dst_addr + 4; // Increment to the next location
end

`uvm_info("run:", $sformatf("Finished transfer end addresses SRC: %0h DST:%0h",
                           src_addr, dst_addr), UVM_LOW)

endtask: body

endclass: mem_trans_seq

// This test shows how to randomize the memory_trans_seq
// to set it up for a block transfer
//
class seq_rand_test extends bus_test_base;

  `uvm_component_utils(seq_rand_test)

```

```

function new(string name = "seq_rand_test", uvm_component parent = null);
    super.new(name);
endfunction

task run_phase( uvm_phase phase );
    phase.raise_objection( this , "start mem_trans_seq" );
    mem_trans_seq seq = mem_trans_seq::type_id::create("seq");

    // Using randomization and constraints to set the initial values
    //
    // This could also be done directly
    //
    assert(seq.randomize() with {src_addr == 32'h0100_0800,
                                dst_addr inside {[32'h0101_0000:32'h0103_0000]};
                                transfer_size == 128;});
    seq.start(m_agent.m_sequencer);
    phase.drop_objection( this , "finished mem_trans_seq" );
endtask: run

endclass: seq_rand_test

```

A SystemVerilog class can randomize itself using this.randomize(), this means that a sequence can re-randomize itself in a loop.

( [download source code examples online at http://verificationacademy.com/uvm-ovm](http://verificationacademy.com/uvm-ovm) ).

## Sequence Object Persistance

When a sequence is created and then executed using sequence.start(), the sequences body method is executed. When the body method completes, the sequence object is still present in memory. This means that any information contained within the sequence and its object hierarchy is still accessible. This feature can be exploited to chain a series of sequences together, using the information from one sequence to seed the execution of another.

Taking the previous example of the memory transfer sequence, the same sequence could be re-executed without randomization to do a series of sequential transfers of the same size, before re-randomizing the sequence to do a transfer of a different size from a different start location.

Another example is a sequence that does a read or a block of reads from a peripheral. The next sequence can then use the content of the previous sequences data fields to guide what it does.

```

// This class shows how to reuse the values persistent within a sequence
// It runs the mem_trans_seq once with randomized values and then repeats it
// several times without further randomization until the memory limit is
// reached. This shows how the end address values are reused on each repeat.
//
class rpt_mem_trans_seq extends bus_seq_base;

```

```

`uvm_object_utils(rpt_mem_trans_seq)

function new(string name = "rpt_mem_trans_seq");
    super.new(name);
endfunction

task body();
    mem_trans_seq trans_seq = mem_trans_seq::type_id::create("trans_seq");

    // First transfer:
    assert(trans_seq.randomize() with {src_addr inside {[32'h0100_0000:32'h0100_FFFF]};
                                         dst_addr inside {[32'h0103_0000:(32'h0104_0000 - (transfer_size*4))]};
                                         transfer_size < 512;
                                         solve transfer_size before dst_addr;});

    trans_seq.start(m_sequencer);
    // Continue with next block whilst we can complete within range
    // Each block transfer continues from where the last one left off
    while ((trans_seq.dst_addr + (trans_seq.transfer_size*4)) < 32'h0104_0000) begin
        trans_seq.start(m_sequencer);
    end

endtask: body

endclass: rpt_mem_trans_seq

```

( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

## Exploiting Sequence Polymorphism

If a library of sequences is created, all of which are derived from the same object type, then it is possible to create these and put them into an array and then execute them in a random order. That order can be made random either by randomly generating the index to the array, or by shuffling the order of the array using the <array>.shuffle() method.

```

// This sequence executes some sub-sequences in a random order
//
class rand_order_seq extends bus_seq_base;

`uvm_object_utils(rand_order_seq)

function new(string name = "");
    super.new(name);
endfunction

//

```

```
// The sub-sequences are created and put into an array of
// the common base type.
//
// Then the array order is shuffled before each sequence is
// randomized and then executed
//
task body;
    bus_seq_base seq_array[4];

    seq_array[0] = n_m_rw_interleaved_seq::type_id::create("seq_0");
    seq_array[1] = rwr_seq::type_id::create("seq_1");
    seq_array[2] = n_m_rw_seq::type_id::create("seq_2");
    seq_array[3] = fill_memory_seq::type_id::create("seq_3");

    // Shuffle the array contents into a random order:
    seq_array.shuffle();
    // Execute all the array items in turn
    foreach(seq_array[i]) begin
        if(!seq_array[i].randomize()) begin
            `uvm_error("body", "randomization failed for req")
        end
        seq_array[i].start(m_sequencer);
    end

    endtask: body

endclass: rand_order_seq
```

Sequences can also be overridden with sequences of a derived type using the UVM factory, see the article on **overriding sequences** for more information. This approach allows a generation flow to change its characteristics without having to change the original sequence code.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Sequences/Overrides

Sometimes, during stimulus generation, it is useful to change the behaviour of sequences or sequence items. The UVM factory provides an override mechanism to be able to substitute one object for another without changing any testbench code and without having to recompile it.

The UVM factory allows factory registered objects to be overridden by objects of a derived type. This means that when an object is constructed using the `<class_name>::type_id::create()` approach, then a change in the factory lookup for that object results in a pointer to an object of a derived type being returned. For instance, if there is sequence of type `seq_a`, and this is extended to create a sequence of type `seq_b` then `seq_b` can be used to override `seq_a`.

There are two types of factory override available - a type override, and an instance override.

## Sequence Type Factory Override:

A type override means that any time a specific object type is constructed using the factory, a handle to the overridden type is returned. A type override can be used with a sequence, and it should be part of the test case configuration in the test. Once the type factory override is set, it will apply to all places in the subsequent sequence code where the overridden sequence object is constructed.

## Sequence Instance Factory Override:

Specific sequences can be overridden via their "path" in the UVM testbench component hierarchy. For `uvm_components`, the path is defined as part of the build process via the name and parent arguments to the `create` method. However, sequences are `uvm_objects` and only use a name argument in their constructor and are not linked into the `uvm_component` hierarchy. The solution for creating a path for a sequence is to use two further arguments to the `create` method. The third argument passed to the sequence can be populated by the results of a `get_full_name()` call, or it can be any arbitrary string. The instance override then uses this string concatenated with the instance name field of the sequence to recreate the "instance path" of the sequence. For obvious reasons this means that a sequence instance override has to be pre-meditated as part of a sequences architecture.

```

//  

// The build method of a test class:  

//  

// Inheritance:  

//  

// a_seq <- b_seq <- c_seq  

//  

function void build_phase( uvm_phase phase );  

  m_env = sot_env::type_id::create("m_env", this);  

  // Set type override  

  b_seq::type_id::set_type_override(c_seq::get_type());  

  // Set instance override - Note the "path" argument see the line for s_a creation  

  // in the run method  

  a_seq::type_id::set_inst_override(c_seq::get_type(), "bob.s_a");  

endfunction: build

```

```
//  
// Run method  
//  
task run_phase( uvm_phase phase );  
  a_seq s_a; // Base type  
  b_seq s_b; // b_seq extends a_seq  
  c_seq s_c; // c_seq extends b_seq  
  
  phase.raise_objection( this , "start a,b and c sequences" );  
  
  // Instance name is "s_a" - first argument,  
  // path name is "bob" but is more usually get_full_name() - third argument  
  s_a = a_seq::type_id::create("s_a",, "bob");  
  // More usual create call  
  s_b = b_seq::type_id::create("s_b");  
  s_c = c_seq::type_id::create("s_c");  
  
  s_a.start(m_env.m_a_agent.m_sequencer); // Results in c_seq being executed  
  s_b.start(m_env.m_a_agent.m_sequencer); // Results in c_seq being executed  
  s_c.start(m_env.m_a_agent.m_sequencer);  
  
  phase.drop_objection( this , "a,b and c sequences done" );  
  
endtask: run
```

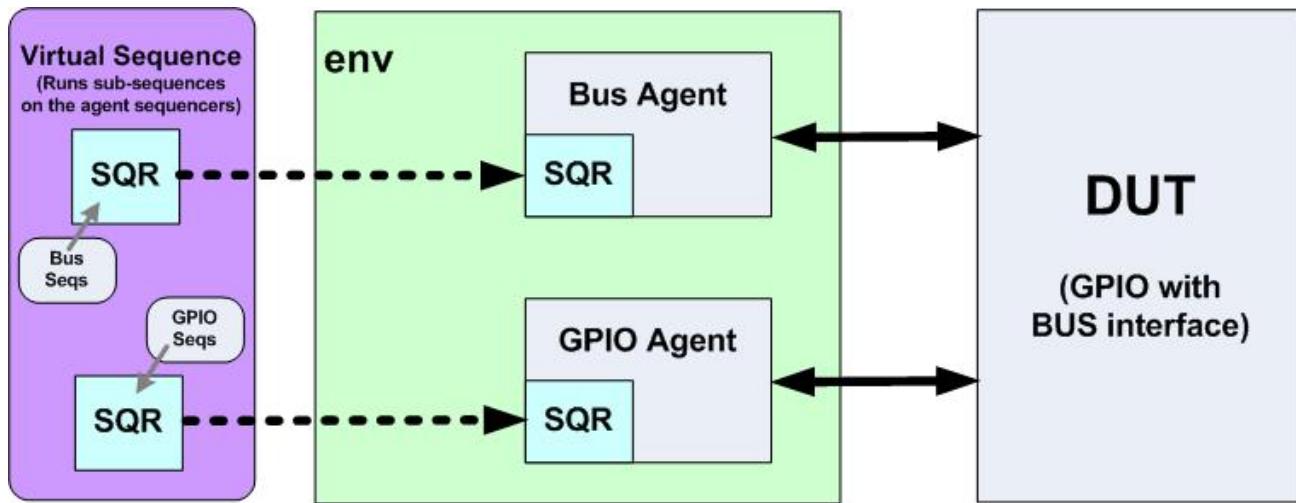
( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

### **Sequence Item Overrides:**

In principle, the same factory override mechanism could be used for sequence\_items. However, it would require a driver to be written with prior knowledge of the derived type, so that it could cast a received sequence\_item to the right type. Therefore, from a practical point of view, sequence\_items are unlikely to be overridden.

# Sequences/Virtual

A virtual sequence is a sequence which controls stimulus generation using several sequencers. Since sequences, sequencers and drivers are focused on point interfaces, almost all testbenches require a virtual sequence to co-ordinate the stimulus across different interfaces and the interactions between them. A virtual sequence is often the top level of the **sequence hierarchy**. A virtual sequence might also be referred to as a 'master sequence' or a 'co-ordinator sequence'.



**The Virtual Sequence**

A virtual sequence differs from a normal sequence in that its primary purpose is not to send sequence items. Instead, it generates and executes sequences on different target agents. To do this it contains handles for the target sequencers and these are used when the sequences are started.

```
// Creating a useful virtual sequence type:
typedef uvm_sequence #(uvm_sequence_item) uvm_virtual_sequence;

// Virtual sequence example:
class my_vseq extends uvm_virtual_sequence;
  ...
  // Handles for the target sequencers:
  a_sequencer_t a_sequencer;
  b_sequencer_t b_sequencer;

  task body();
    ...
    // Start interface specific sequences on the appropriate target sequencers:
    aseq.start( a_sequencer , this );
    bseq.start( b_sequencer , this );
  endtask
endclass
```

In order for the virtual sequence to work, the sequencer handles have to be assigned. Typically, a virtual sequence is created in a test class in the run phase and the assignments to the sequencer handles within the virtual sequence object are

made by the test. Once the sequencer handles are assigned, the virtual sequence is started using a null for the sequencer handle.

```
my_seq vseq = my_seq::type_id::create("vseq");

vseq.a_sequencer = env.subenv1.bus_agent.sequencer;
vseq.b_sequencer = env.subenv2.subsubenv1.bus_agent3.sequencer;

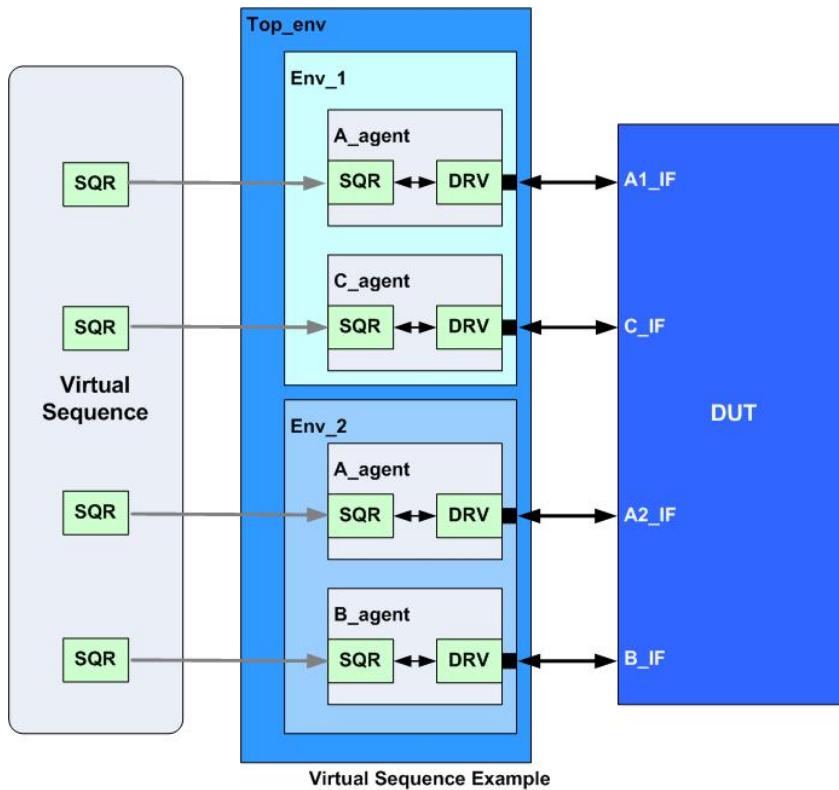
vseq.start( null );
```

There are several variations on the virtual sequence theme. There is nothing to stop the virtual sequence being started on a sequencer and sending sequence items to that sequencer whilst also executing other sequences on their target sequencers. The virtual sequence does not have to be executed by the test, it may be executed by an environment encapsulating a number of agents. For a large testbench with many agents and several areas of concern there may be several virtual sequences running concurrently.

In addition to target sequencer handles, a virtual sequence may also contain handles to other testbench resources such as **register models** which would be used by the sub-sequences.

## Recommended Virtual Sequence Initialisation Methodology

In order to use the (U)OVM effectively, many organisations separate the implementation of the testbench from the implementation of the test cases. This is either a conceptual separation or a organisational separation. The testbench implementor should provide a test base class and a base virtual sequence class from which test cases can be derived. The test base class is responsible for building and configuring the verification environment component hierarchy, and specifying which virtual sequence(s) will run. The test base class should also contain a method for assigning sequence handles to virtual sequences derived from the virtual sequence base class. With several layers of vertical reuse, the hierarchical paths to target sequencers can become quite long. Since the hierarchical paths to the target sequencers are known to the testbench writer, this information can be encapsulated for all future test case writers.



As an example consider the testbench illustrated in the diagram. To illustrate a degree of virtual reuse, there are four target agents organised in two sub-environments within a top-level environment. The virtual sequence base class contains handles for each of the target sequencers:

```
class top_vseq_base extends uvm_sequence #(uvm_sequence_item);

`uvm_object_utils(top_vseq_base)

uvm_sequencer #(a_seq_item) A1;
uvm_sequencer #(a_seq_item) A2;
uvm_sequencer #(b_seq_item) B;
uvm_sequencer #(c_seq_item) C;

function new(string name = "top_vseq_base");
    super.new(name);
endfunction

endclass: top_vseq_base
```

In the test base class a method is created which can be used to assign the sequencer handles to the handles in classes derived from the virtual sequence base class.

```
class test_top_base extends uvm_test;

`uvm_component_utils(test_top_base)
```

```

env_top m_env;

function new(string name = "test_top_base", uvm_component parent = null);
    super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
    m_env = env_top::type_id::create("m_env", this);
endfunction: build_phase

// Method to initialise the virtual sequence handles
function void init_vseq(top_vseq_base vseq);
    vseq.A1 = m_env.m_env_1.m_agent_a.m_sequencer;
    vseq.C = m_env.m_env_1.m_agent_c.m_sequencer;
    vseq.A2 = m_env.m_env_2.m_agent_a.m_sequencer;
    vseq.B = m_env.m_env_2.m_agent_b.m_sequencer;
endfunction: init_vseq

endclass: test_top_base

```

In a test case derived from the test base class the virtual sequence initialisation method is called before the virtual sequence is started.

```

class init_vseq_from_test extends test_top_base;

`uvm_component_utils(init_vseq_from_test)

function new(string name = "init_vseq_from_test", uvm_component parent = null);
    super.new(name, parent);
endfunction

task run_phase(uvm_phase phase);
    vseq_A1_B_C vseq = vseq_A1_B_C::type_id::create("vseq");

    phase.raise_objection(this);

    init_vseq(vseq); // Using method from test base class to assign sequence handles
    vseq.start(null); // null because no target sequencer

    phase.drop_objection(this);
endtask: run_phase

endclass: init_vseq_from_test

```

The virtual sequence is derived from the virtual sequence base class and requires no initialisation code.

```

class vseq_A1_B_C extends top_vseq_base;

`uvm_object_utils(vseq_A1_B_C)

function new(string name = "vseq_A1_B_C");
    super.new(name);
endfunction

task body();
    a_seq a = a_seq::type_id::create("a");
    b_seq b = b_seq::type_id::create("b");
    c_seq c = c_seq::type_id::create("c");

    a.start(A1);
    fork
        b.start(B);
        c.start(C);
    join

endtask: body

endclass: vseq_A1_B_C

```

This example illustrates how the target sequencer handles can be assigned from the test case, but the same approach could be used for passing handles to other testbench resources such as register models and configuration objects which may be relevant to the operation of the virtual sequence or its sub-sequences.

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

## Alternative Methods For Initialising A Virtual Sequence

In the preceding example, the virtual sequence is initialised from the test class, this is the recommended approach since it is so simple to implement. However, there are several alternative approaches that could be used:

- Putting the initialisation method in the test package rather than making it a member of a test base class. This allows several variants of the init method to exist without having to define different base classes.
- Putting the initialisation method in a mapping package which imports the env package(s) and the virtual sequence package. This separates out the initialisation from the test. Several initialisation methods could be defined for different virtual sequences. The mapping package would be imported into the test package.

```

package my_virtual_sequence_mapping_pkg;
//
// This package is specific to the test env and to the virtual sequence
//
import my_sequence_pkg::*;
import my_env_pkg::*;

```

```

function void init_my_virtual_sequence_from_my_env( my_virtual_sequence vseq , my_env env );
  vseq.fabric_ports[0] = env.env1.a_agent.sequencer;
  vseq.fabric_ports[1] = env.env2.a_agent.sequencer;
  vseq.data_port = env.env1.b_agent.sequencer;
  vseq.control_port = env.env2.c_agent.sequencer;
end

// Other virtual sequence initialisation methods could also be defined

endpackage

```

- Using the `uvm_config_db` to pass sequencer handles from the env to the virtual sequence. This can be made to work in small scaled environments, but may breakdown in larger scale environments, especially when multiple instantiations of the same env are used since there is no way to uniquify the look-up key for the sequencer in the `uvm_config_db`

```

// Inside the env containing the target sequencers:
// 

function void connect_phase(uvm_phase phase);
// 

  uvm_config_db #(a_sequencer)::set(null, "Sequencers", "a_sqr", a_agent.m_sequencer);
  uvm_config_db #(b_sequencer)::set(null, "Sequencers", "b_sqr", b_agent.m_sequencer);
// 

endfunction

// Inside the virtual sequence base class:
// 

a_sequencer A;
b_sequencer B;

// Get the sequencer handles back from the config_db
// 

task body();
  if(!uvm_config_db #(a_sequencer)::get(null, "Sequencers", "a_sqr", A)) begin
    `uvm_error("body", "a_sqr of type a_sequencer not found in the uvm_config_db")
  end
  if(!uvm_config_db #(b_sequencer)::get(null, "Sequencers", "b_sqr", B)) begin
    `uvm_error("body", "b_sqr of type b_sequencer not found in the uvm_config_db")
  end
  // ...
endtask

```

- Using the `find_all()` method to find all the sequencers that match a search string in an environment. Again this relies on the sequencer paths being unique which is an assumption that will most likely break down in larger scale environments.

```
//  
// A virtual sequence which runs stand-alone, but finds its own sequencers  
class virtual_sequence_base extends uvm_sequence #(uvm_sequence_item);  
  
`uvm_object_utils(virtual_sequence)  
  
// Sub-Sequencer handles  
bus_sequencer_a A;  
gpio_sequencer_b B;  
  
// This task would be called as super.body by inheriting classes  
task body;  
    get_sequencers();  
endtask: body  
  
//  
function void get_sequencers;  
    uvm_component tmp[$];  
    //find the A sequencer in the testbench  
    tmp.delete(); //Make sure the queue is empty  
    uvm_top.find_all("*m_bus_agent_h.m_sequencer_h", tmp);  
    if (tmp.size() == 0)  
        `uvm_fatal(report_id, "Failed to find mem sequencer")  
    else if (tmp.size() > 1)  
        `uvm_fatal(report_id, "Matched too many components when looking for mem sequencer")  
    else  
        $cast(A, tmp[0]);  
    //find the B sequencer in the testbench  
    tmp.delete(); //Make sure the queue is empty  
    uvm_top.find_all("*m_gpio_agent_h.m_sequencer_h", tmp);  
    if (tmp.size() == 0)  
        `uvm_fatal(report_id, "Failed to find mem sequencer")  
    else if (tmp.size() > 1)  
        `uvm_fatal(report_id, "Matched too many components when looking for mem sequencer")  
    else  
        $cast(B, tmp[0]);  
endfunction: get_sequences  
  
endclass: virtual_sequence_base
```

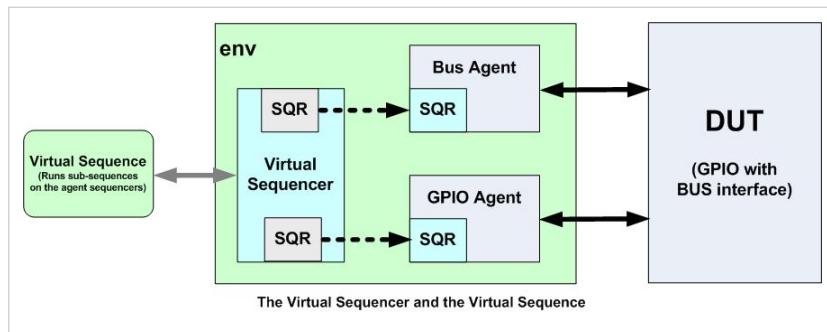
## The Virtual Sequencer - An Alternative Methodology For Running Virtual Sequences

An alternative methodology for running virtual sequences is to use a **virtual sequencer**, which is a uvm\_sequencer which contains the handles for the target sequencers. In this methodology, the virtual sequence is started on the virtual sequencer and it gets the handles for the target sequencers from the virtual sequencer. The limitation of this approach is that it is a fixed implementation which is very tightly coupled to the local hierarchy of an env and this adds complications with vertical reuse.

---

# Sequences/VirtualSequencer

A virtual sequence is a sequence which controls a stimulus generation process using several sequencers. Since sequences, sequencers and drivers are focused on interfaces, almost all testbenches require a virtual sequence to co-ordinate the stimulus across different interfaces and the interactions between them.



A virtual sequence can be implemented in one of two ways, the recommended way is to use a stand-alone **virtual sequence** and the 'legacy' alternative is to use virtual sequence that is designed to run on a virtual sequencer as described here.

A virtual sequencer is a sequencer that is not connected to a driver itself, but contains handles for sequencers in the testbench hierarchy.

## Virtual sequences that run on virtual sequencers

A virtual sequence is designed to run on a virtual sequencer by adding code that gets the handles to the sub-sequencers from the virtual sequencer. The most convenient way to do this is to extend virtual sequences from a base class that does the sequencer handle assignment. The virtual sequencer is part of the UVM component hierarchy and so its sub-sequencer references can be assigned during the connect phase.

Typically, a virtual sequencer is inserted inside the env of a block level testbench, with the connect method of the block level env being used to assign the sub-sequencer handles. Since the virtual\_sequence is likely to be in overall control of the simulation, it is usually created in the test run method and started on the virtual sequencer - i.e. `virtual_sequence.start(virtual_sequencer);`

A useful coding guideline is to give the sub-sequencers inside the virtual sequence a name that represents the interface that they are associated with, this makes life easier for the test writer. For instance, the sequencer for the master bus interface could be called "bus\_master" rather than "master\_axi\_sequencer".

```
// Virtual sequencer class:
class virtual_sequencer extends uvm_virtual_sequencer;

  `uvm_component_utils(virtual_sequencer)

  // Note that the handles are in terms that the test writer understands
  bus_master_sequencer bus;
  gpio_sequencer gpio;

  function new(string name = "virtual_sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction
```

```
endclass: virtual_sequencer

class env extends uvm_env;

// Relevant parts of the env which combines the
// virtual_sequencer and the bus and gpio agents
//
// Build:
function void build_phase( uvm_phase phase );
    m_bus_agent = bus_master_agent::type_id::create("m_bus_agent", this);
    m_gpio_agent = gpio_agent::type_id::create("m_gpio_agent", this);
    m_v_sqr = virtual_sequencer::type_id::create("m_v_sqr", this);
endfunction: build_phase

// Connect - where the virtual_sequencer is hooked up:
// Note that these references are constant in the context of this env
function void connect_phase( uvm_phase phase );
    m_v_sqr.bus = m_bus_agent.m_sequencer;
    m_v_sqr gpio = m_gpio_agent.m_sequencer;
endfunction: connect_phase

endclass:env

// Virtual sequence base class:
//
class virtual_sequence_base extends uvm_sequence #(uvm_sequence_item);

`uvm_object_utils(virtual_sequence_base)

// This is needed to get to the sub-sequencers in the
// m_sequencer
virtual_sequencer v_sqr;

// Local sub-sequencer handles
bus_master_sequencer bus;
gpio_sequencer gpio;

function new(string name = "virtual_sequence_base");
    super.new(name);
endfunction

// Assign pointers to the sub-sequences in the base body method:
task body();
    if(!$cast(v_sqr, m_sequencer)) begin
```

```
 `uvm_error(get_full_name(), "Virtual sequencer pointer cast failed");
end

bus = v_sqr.bus;
gpio = v_sqr/gpio;
endtask: body

endclass: virtual_sequence_base

// Virtual sequence class:
// 
class example_virtual_seq extends virtual_sequence_base;

random_bus_seq bus_seq;
random_gpio_chunk_seq gpio_seq;

`uvm_object_utils(example_virtual_seq)

function new(string name = "example_virtual_seq");
    super.new(name);
endfunction

task body();
    super.body; // Sets up the sub-sequencer pointers
    gpio_seq = random_gpio_chunk_seq::type_id::create("gpio_seq");
    bus_seq = random_bus_seq::type_id::create("bus_seq");

    repeat(20) begin
        bus_seq.start(bus);
        gpio_seq.start(gpio);
    end
endtask: body

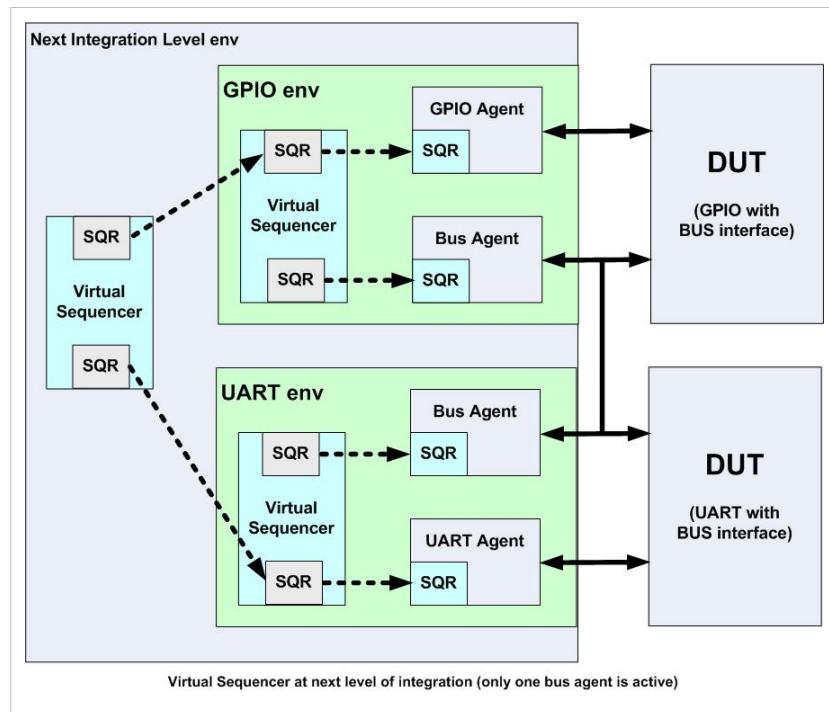
endclass: example_virtual_seq

// Inside the test class:
// 
task run;
    example_virtual_sequence test_seq = example_virtual_sequencer::type_id::create("test_seq");

    //...
    test_seq.start(m_env.m_v_sqr);
    //...
endtask: run
```

## Chaining Virtual Sequencers for vertical reuse

When creating integrated verification environments that reuse block level envs, the virtual sequencers at the different levels of the testbench hierarchy can be chained together, allowing virtual sequences or sub-sequences to be run at any level. To achieve this level of flexibility, both the sub-sequencer handles and the virtual sequencer handles need to be encapsulated at each successive level.



```
// Virtual sequencer from the UART env
class uart_env_virtual_sqr extends uvm_virtual_sequencer;
// ...
uart_sequencer uart;
bus_sequencer bus;
// ...
endclass: uart_env_virtual_sqr

// Virtual sequencer from the GPIO env
class gpio_env_virtual_sqr extends uvm_virtual_sequencer;
// ...
gpio_sequencer gpio;
bus_sequencer bus;
// ...
endclass: gpio_env_virtual_sqr

// Virtual sequencer from the SoC env
class soc_env_virtual_sqr extends uvm_virtual_sequencer;
//...
// Low level sequencers to support virtual sequences running across
// TB hierarchical boundaries
uart_sequencer uart;
bus_sequencer uart_bus;
```

```

gpio_sequencer gpio;
bus_sequencer gpio_bus;

// Virtual sequencers to support existing virtual sequences
//
uart_env_virtual_sqr uart_v_sqr;
gpio_env_virtual_sqr gpio_v_sqr;

// Low level sequencer pointer assignment:
// This has to be after connect because these connections are
// one down in the hierarchy
function void end_of_elaboration();
    uart = uart_v_sqr.uart;
    uart_bus = uart_v_sqr.bus;
    gpio = gpio_v_sqr gpio;
    gpio_bus = gpio_v_sqr.bus;
endfunction: end_of_elaboration

endclass: soc_env_virtual_sqr

```

### Coding Guideline: Virtual Sequences should check for null sequencer pointers before executing

Virtual sequence implementations are based on the assumption that they can run sequences on sub-sequencers within agents. However, agents may be passive or active depending on the configuration of the testbench. In order to prevent test cases crashing with null handle errors, virtual sequences should check that all of the sequencers that they intend to use have valid handles. If a null sequencer handle is detected, then they should bring the test case to an end with an `uvm_report_fatal` call.

```

// Either inside the virtual sequence base class or in
// an extension of it that will use specific sequencers:
task body();
    if(!$cast(v_sqr, m_sequencer)) begin
        `uvm_error(get_full_name(), "Virtual sequencer pointer cast failed")
    end
    if(v_sqr gpio == null) begin
        `uvm_fatal(get_full_name(), "GPIO sub-sequencer null pointer: this test case will fail, check config or virtual sequence")
    end
    else begin
        gpio = v_sqr gpio;
    end
    if(v_sqr gpio_bus == null) begin
        `uvm_fatal(get_full_name(), "BUS sub-sequencer null pointer: this test case will fail, check config or virtual sequence")
    end
    else begin

```

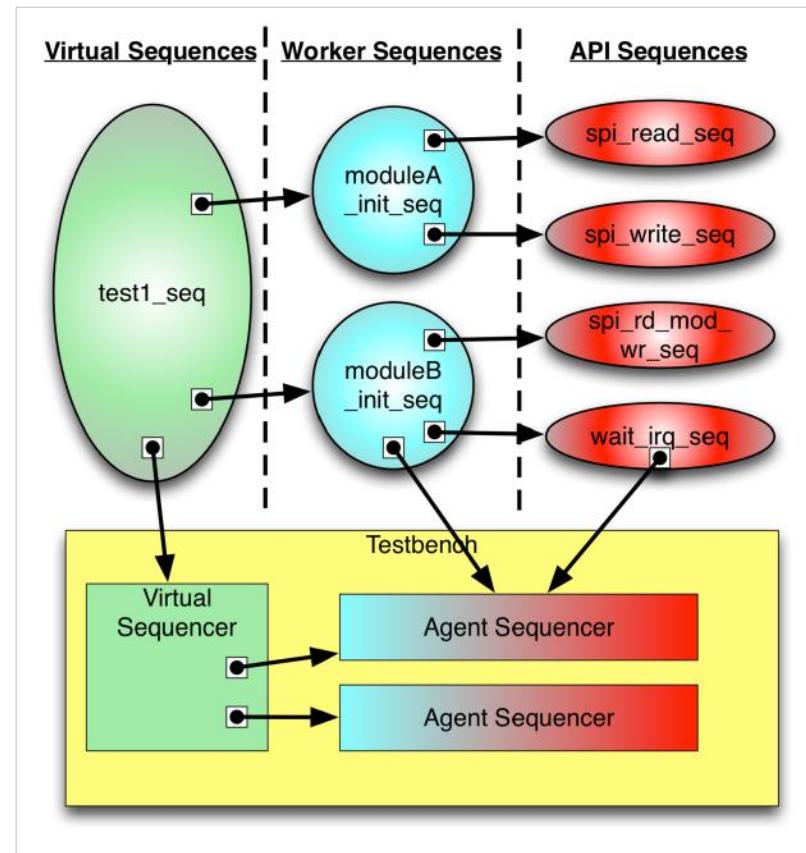
```
  gpio_bus = v_sqr/gpio_bus;  
end  
endtask: body
```

# Sequences/Hierarchy

When dealing with sequences, it helps to think in layers when considering the different functions that a testbench will be asked to perform. At the lowest layer associated with each agent are API sequences. The middle layer which makes use of the API sequences to get work done are worker sequences. Finally at the top of the testbench controlling everything is a **virtual sequence** which co-ordinates the running of worker sequences on the different target sequencers.

## API Sequences

API sequences are the lowest layer in the sequence hierarchy. They also do the least amount of work in that they are doing a very targeted action when they are run. API sequences should perform small discreet actions like performing a read on a bus, a write on a bus, a read-modify-write on a bus or waiting for an interrupt or some other signal. The API sequences would be included in the SystemVerilog package which defines the Agent on which the sequences are intended to run. Two example API sequences would look as follows:



```
//API Sequence for doing a Read
class spi_read_seq extends uvm_sequence #(spi_item);
  `uvm_object_utils(spi_read_seq)

  const string      report_id = "spi_read_seq";
  rand  bit [7:0]   addr;
  bit [15:0]      rdata;

  task body();
    req = spi_item::type_id::create("spi_request");
    start_item(req);
    if ( !(req.randomize() with {req.addr == local::addr;}) ) {
      `uvm_error(report_id, "Randomize Failed!")
    finish_item(req);
    rdata = req.data;
  end
endclass
```

```
  endtask : body

  task read(input bit [7:0] addr, output bit [15:0] read_data,
            input uvm_sequencer_base seqr, input uvm_sequence_base parent = null);
    this.addr = addr;
    this.start(seqr, parent);
    read_data = req.data;
  endtask : read

endclass : spi_read_seq

//API Sequence for doing a Write
class spi_write_seq extends uvm_sequence #(spi_item);
  `uvm_object_utils(spi_write_seq)

  const string      report_id = "spi_write_seq";
  rand  bit [7:0]   addr;
  rand  bit [15:0]  wdata;

  task body();
    req = spi_item::type_id::create("spi_request");
    start_item(req);
    if ( !(req.randomize() with {req.addr == local::addr;
                                req.data == local::wdata; } )) {
      `uvm_error(report_id, "Randomize Failed!")
    finish_item(req);
  endtask : body

  task write(bit [7:0] addr, bit [15:0] write_data,
            uvm_sequencer_base seqr, uvm_sequence_base parent = null);
    this.addr = addr;
    this.wdata = write_data;
    this.start(seqr, parent);
  endtask : write

endclass : spi_write_seq
```

## Worker Sequences

Worker sequences make use of the low level API sequences to build up middle level sequences. These mid-level sequences could do things such as dut configuration, loading a memory, etc. Usually a worker sequence would only be sending sequence items to a single sequencer. A worker sequence would look like this:

```
//Worker sequence for doing initial configuration for Module A
class moduleA_init_seq extends uvm_sequence #(spi_item);
  `uvm_object_utils(moduleA_init_seq)

  const string          report_id = "moduleA_init_seq";
  spi_read_seq         read;
  spi_write_seq        write;

  task body();
    read = spi_read_seq::type_id::create("read");
    write = spi_write_seq::type_id::create("write");

    //Configure registers in Module
    //Calling start
    write.addr = 8'h20;
    write.wdata = 16'h00ff;
    write.start(m_sequencer, this);

    //Using the write task
    write.write(8'h22, 16'h0100, m_sequencer, this);

    //Other register writes

    //Check that Module A is ready
    read.addr = 8'h2c;
    read.start(m_sequencer, this);
    if (read.rdata != 16'h0001)
      `uvm_fatal(report_id, "Module A is not ready")

  endtask : body

endclass : moduleA_init_seq
```

## Virtual Sequences

Virtual sequences are used to call and coordinate all the worker sequences. In most cases, designs will need to be initialized before random data can be sent in. The virtual sequence can call the worker initialization sequences and then call other worker sequences or even API sequences if it needs to do a low level action. The virtual sequence will either contain handles to the target sequencers (recommended) or be running on a **virtual sequencer** which allows access to all of the sequencers that are needed to run the worker and API sequences. An example virtual sequence would look like this:

```
//Virtual Sequence controlling everything
class test1_seq extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(test1_seq)

  const string          report_id = "test1_seq";

  // These handles will be assigned by an init method in the test
  uvm_sequencer_base    spi_seqr;
  uvm_sequencer_base    modA_seqr;
  uvm_sequencer_base    modB_seqr;

  moduleA_init_seq      modA_init;
  moduleB_init_seq      modB_init;

  moduleA_rand_data_seq modA_rand_data;
  moduleB_rand_data_seq modB_rand_data;

  spi_read_seq          spi_read;
  bit [15:0]             read_data;

  task body();
    modA_init = moduleA_init_seq::type_id::create("modA_init");
    modB_init = moduleB_init_seq::type_id::create("modB_init");

    modA_rand_data = moduleA_rand_data_seq::type_id::create("modA_rand_data");
    modB_rand_data = moduleB_rand_data_seq::type_id::create("modB_rand_data");

    spi_read = spi_read_seq::type_id::create("spi_read");

    //Do Initial Config
    fork
      modA_init.start(spi_seqr, this);
      modB_init.start(spi_seqr, this);
    join
    //Now start random data (These would probably be started on different sequencers for a real design)
  endtask
endclass
```

```
fork
  modA_rand_data.start(modA_seqr, this);
  modB_rand_data.start(modB_seqr, this);
join

//Do a single read to check completion
spi_read.read(8'h7C, read_data, spi_seqr, this);
if (read_data != 16'hffff)
  `uvm_error(report_id, "Test Failed!")

endtask : body

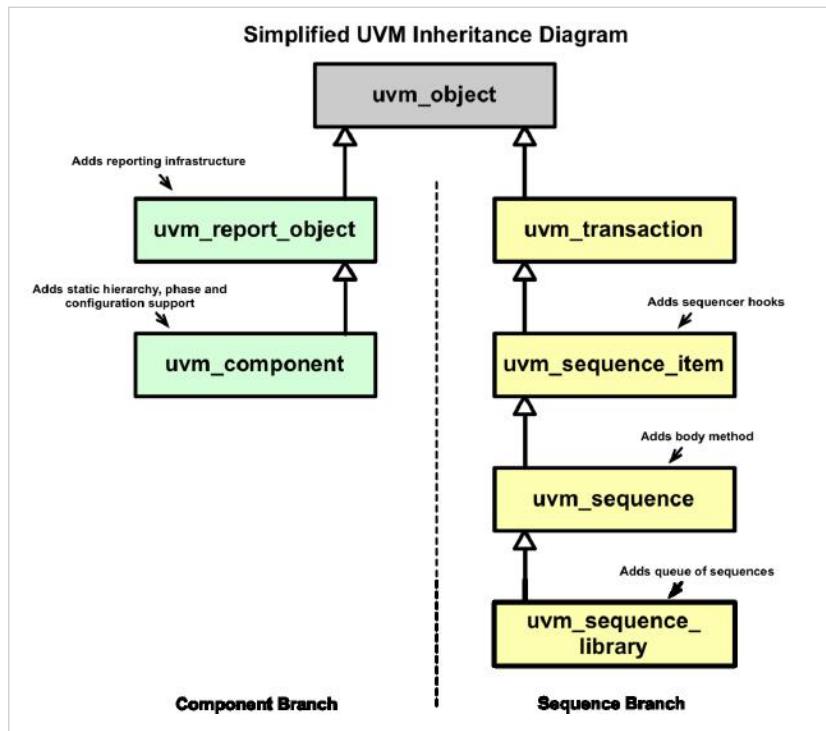
endclass : test1_seq
```

# Sequences/SequenceLibrary

UVM provides a class for randomly creating and running sequences. This class is called `uvm_sequence_library`. The `uvm_sequence_library` class inherits from `uvm_sequence` which means that an instance of a sequence library is also a sequence. This is important because after a sequence library is configured, it is used just as a sequence is used.

Whilst the sequence library is available to use, it is not the preferred method for creating and starting sequences and it is not recommended. The techniques described in the Sequences/Generation article generally should be used as those techniques provide easy control over serial and parallel execution of sequences. The techniques described on the Sequences/Generation page also allows for the maximum reuse of a test stimulus.

The `uvm_sequence_library` class is replacing the sequencer library functionality which is deprecated in UVM. The sequencer library was used by some in OVM, but it had several limitations.



## Functionality

The `uvm_sequence_library` class provides a means for randomly selecting a sequence to run from a list of sequences registered with it. The total number of sequences run when the sequence library is started is controllable with a default of 10 (as of UVM 1.1a).

## Basics

To create a sequence library, a class is extended from the parameterized `uvm_sequence_library` class. There are two parameter values which can potentially be set. They are the same REQ and RSP parameter values that a sequence requires. Also like a sequence, the RSP parameter defaults to having the same value as the REQ parameter.

After extending from the `uvm_sequence_library` class, then the standard factory registration takes place using the ``uvm_object_utils()` macro. Then a unique ``uvm_sequence_library_utils()` macro needs to be invoked. This macro invocation is paired with a call to the function `init_sequence_library()`. The macro and function are needed to populate the sequence library with any sequences that were statically registered with it or any of its base classes.

```

class mem_seq_lib extends uvm_sequence_library #(mem_item);
  `uvm_object_utils(mem_seq_lib)
  `uvm_sequence_library_utils(mem_seq_lib)
  
```

```

function new(string name="mem_seq_lib");
    super.new(name);
    init_sequence_library();
endfunction : new

endclass : mem_seq_lib

```

## Registering Sequences

Sequences can be registered with a sequence library either by using a macro or function calls. Mentor Graphics recommends using function calls as they are more flexible and do not require extra code inside of standard sequences. The macro method requires adding a macro call ( `uvm\_add\_to\_seq\_lib(<sequence\_type>, <seq\_library\_type>) ) to standard sequences which may already exist or which may have been written by someone else.

To register a sequence(s) with every instance of a particular type of sequence library, the add\_typewide\_sequence() and/or add\_typewide\_sequences() functions are used.

```

class mem_seq_lib extends uvm_sequence_library #(mem_item);
    ...

    function new(string name="mem_seq_lib");
        super.new(name);

        //Explicitly add the memory sequences to the library
        add_typewide_sequences({mem_seq1::get_type(), mem_seq2::get_type(), mem_seq3::get_type(),
                               mem_seq4::get_type(), mem_seq5::get_type(), mem_seq6::get_type()});

        init_sequence_library();
    endfunction : new

endclass : mem_seq_lib

```

The most convenient location to place the add\_typewide\_sequence() and/or add\_typewide\_sequences() call is in the constructor of the sequence\_library.

Sequences can also be registered with individual instances of a sequence library by using the add\_sequence() or the add\_sequences() function. This typically would be done in the test where the sequence is instantiated.

```

class test_seq_lib extends test_base;
    ...

    task main_phase(uvm_phase phase);
        phase.raise_objection(this, "Raising Main Objection");

        //Register another sequence with this sequence library instance
        seq_lib.add_sequence( mem_error_seq::get_type() );
    endtask

```

```
//Start the mem sequence
seq_lib.start(m_mem_sequencer); //This task call is blocking

phase.drop_objection(this, "Dropping Main Objection");
endtask : main_phase

endclass : test_seq_lib
```

## Controls

The `uvm_sequence_library` class provides some built-in controls to constrain the sequence library when it is `randomized()`. To control the number of sequences which are run when the sequence library is started, the data members `min_random_count` (default of 10) and `max_random_count` (default of 10) can be changed. These values can either be changed in the test where the sequence is instantiated

```
class test_seq_lib extends test_base;

...

task main_phase(uvm_phase phase);
    phase.raise_objection(this, "Raising Main Objection");

    //Configure the constraints for how many sequences are run
    seq_lib.min_random_count = 5;
    seq_lib.max_random_count = 12;

    //Randomize the sequence library
    if (!seq_lib.randomize())
        `uvm_error(report_id, "The mem_seq_lib library failed to randomize()")

    //Start the mem sequence
    seq_lib.start(m_mem_sequencer); //This task call is blocking

    phase.drop_objection(this, "Dropping Main Objection");
endtask : main_phase

endclass : test_seq_lib
```

or by using the `uvm_config_db` and specifying the full path to the sequence library. This second option is only used if the sequence library is configured to be a default sequence for an UVM phase which is not recommended by Mentor Graphics.

The method of selection for the next sequence to be executed when the sequence library is running can also be controlled. Four options exist and are specified in an enumeration.

Enum Value	Description
UVM_SEQ_LIB RAND	Random sequence selection. This is the default.
UVM_SEQ_LIB RANDC	Random cyclic sequence selection
UVM_SEQ_LIB ITEM	Emit only items, no sequence execution
UVM_SEQ_LIB USER	Apply a user-defined random-selection algorithm

To change the randomization method used, the `selection_mode` data member is set before the sequence library is started.

```
class test_seq_lib extends test_base;  
  
...  
  
task main_phase(uvm_phase phase);  
    phase.raise_objection(this, "Raising Main Objection");  
  
    //Change to RANDC mode for selection  
    seq_lib.selection_mode = UVM_SEQ_LIB RANDC;  
  
    //Start the mem sequence  
    seq_lib.start(m_mem_sequencer); //This task call is blocking  
  
    phase.drop_objection(this, "Dropping Main Objection");  
endtask : main_phase  
  
endclass : test_seq_lib
```

# Driver/Use Models

---

Stimulus generation in the UVM relies on a coupling between sequences and drivers. A sequence can only be written when the characteristics of a driver are known, otherwise there is a potential for the sequence or the driver to get into a deadlock waiting for the other to provide an item. This problem can be mitigated for reuse by providing a set of base utility sequences which can be used with the driver and by documenting the behaviour of the driver.

There are a large number of potential stimulus generation use models for the sequence driver combination, however most of these can be characterised by one of the following use models:

## Unidirectional Non-Pipelined

In the unidirectional non-pipelined use model, requests are sent to the driver, but no responses are received back from the driver. The driver itself may use some kind of handshake mechanism as part of its transfer protocol, but the data payload of the transaction is unidirectional.

An example of this type of use model would be an unidirectional communication link such as an ADPCM or a PCM interface.

## Bidirectional Non-Pipelined

In the bidirectional non-pipelined use model, the data transfer is bidirectional with a request sent from a sequence to a driver resulting in a response being returned to the sequence from the driver. The response occurs in lock-step with the request and only one transfer is active at a time.

An example of this type of use model is a simple bus interface such as the AMBA Peripheral Bus (APB).

## Pipelined

In the pipelined use model, the data transfer is bidirectional, but the request phase overlaps the response phase to the previous request. Using pipelining can provide hardware performance advantages, but it complicates the sequence driver use model because requests and responses need to be handled separately.

CPU buses frequently use pipelines and one common example is the AMBA AHB bus.

## Out Of Order Pipelined

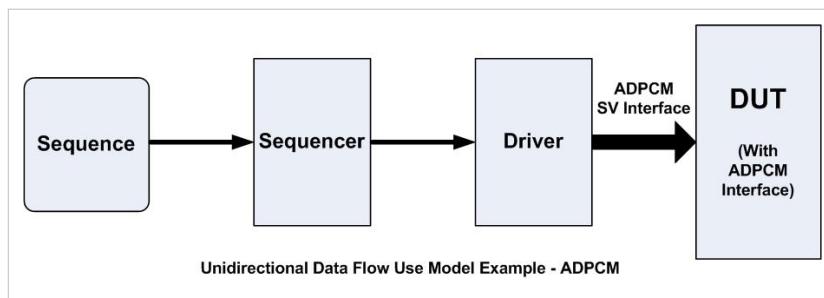
In applications such as fabrics and switches, requests are acknowledged but not responded to in order. The reason being that an accessed resource may already be busy or the path to it may already be in use, and the requesting master has its command accepted leaving it free to issue a new request. This requires a more complicated use model where queues and ids are used to track outstanding responses.

Another variation of this use model would be interleaved bursts where a burst transfer can be interleaved with other transfers.

Advanced SoC buses such as AXI and OCP support out of order responses.

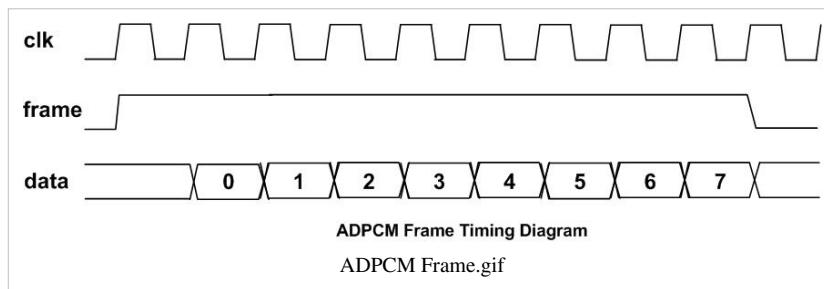
# Driver/Unidirectional

The driver controls the flow of sequence\_items by using `get_next_item()` to obtain the next sequence\_item to be processed, and then does not make the `item_done()` call until it has finished processing the item. The sequence is blocked at its `finish_item()` call until the `item_done()` call is made by the driver.



In the unidirectional non-pipelined sequence driver use model, the data flow is unidirectional. The sequence sends a series of request sequence\_items to the DUT interface but receives no response sequence items. However, the control flow of this use model is bidirectional, since there are handshake mechanisms built into the UVM sequencer communication protocol. The driver may also implement a hand shake on the DUT interface, but this will not be visible to the controlling sequence.

## A Unidirectional Example



An example of a unidirectional dataflow is sending ADPCM packets using a PCM framing protocol. The waveform illustrates the protocol.

```

class adpcm_driver extends uvm_driver #(adpcm_seq_item);

`uvm_component_utils(adpcm_driver)

adpcm_seq_item req;

virtual adpcm_if ADPCM;

function new(string name = "adpcm_driver", uvm_component parent = null);
    super.new(name, parent);
endfunction

task run_phase( uvm_phase phase );
    int top_idx = 0;

```

```

// Default conditions:
ADPCM.frame <= 0;
ADPCM.data <= 0;
forever begin
    seq_item_port.get_next_item(req); // Gets the sequence_item from the sequence
    repeat(req.delay) begin // Delay between packets
        @(posedge ADPCM.clk);
    end

    ADPCM.frame <= 1; // Start of frame
    for(int i = 0; i < 8; i++) begin // Send nibbles
        @(posedge ADPCM.clk);
        ADPCM.data <= req.data[3:0];
        req.data = req.data >> 4;
    end

    ADPCM.frame <= 0; // End of frame
    seq_item_port.item_done(); // Indicates that the sequence_item has been consumed
end
endtask: run

endclass: adpcm_driver

```

The sequence implementation in this case is a loop which generates a series of sequence\_items. A variation on this theme would be for the sequence to actively shape the traffic sent rather than send purely random stimulus.

```

class adpcm_tx_seq extends uvm_sequence #(adpcm_seq_item);

`uvm_object_utils(adpcm_tx_seq)

// ADPCM sequence_item
adpcm_seq_item req;

// Controls the number of request sequence items sent
rand int no_reqs = 10;

function new(string name = "adpcm_tx_seq");
    super.new(name);
endfunction

task body();
    req = adpcm_seq_item::type_id::create("req");

    repeat(no_reqs) begin

```

```
    start_item(req);
    assert(req.randomize());
    finish_item(req);
  end
endtask: body

endclass: adpcm_tx_seq
```

( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

# Driver/Bidirectional

One of the most common form of sequence driver use models is the scenario where the sequencer sends request sequence\_items to the driver which executes the request phase of the pin level protocol, and then the driver responds to the response phase of the pin-level transaction returning the

response back to the sequence. In this use model the flow of data is bidirectional and a new request phase cannot be started until the response phase has completed. An example of this kind of protocol would be a simple peripheral bus such as the AMBA APB.

To illustrate how this use model would be implemented, a DUT containing a GPIO and a bus interface will be used. The bus protocol used is shown in the timing diagram. The request phase of the transaction is initiated by the valid signal becoming active, with the address and direction signal (RNW) indicating which type of bus transfer is taking place. The response phase of the transaction is completed when the ready signal becomes active.

The driver that manages this protocol will collect a request sequence\_item from the sequencer and then drive the bus request phase. The driver waits until the interface ready line becomes active and then returns the response information, which would consist of the error bit and the read data if a read has just taken place.

The recommended way of implementing the driver is to use `get_next_item()` followed by `item_done()` as per the following example:

```
class bidirect_bus_driver extends uvm_driver #(bus_seq_item);

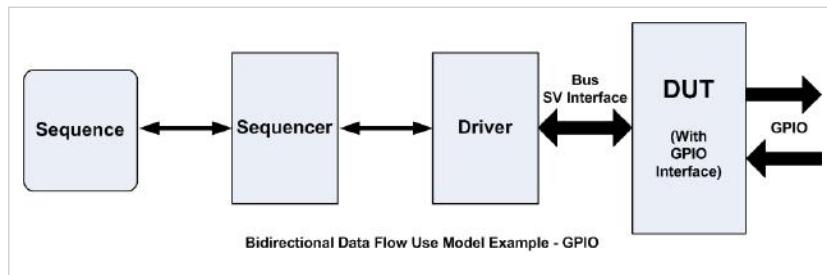
`uvm_component_utils(bidirect_bus_driver)

bus_seq_item req;

virtual bus_if BUS;

function new(string name = "bidirect_bus_driver", uvm_component parent = null);
  super.new(name, parent);
endfunction

task run_phase( uvm_phase phase );
  // Default conditions:
  BUS.valid <= 0;
  BUS.rnw <= 1;
  // Wait for reset to end
  @(posedge BUS.resetn);
  forever begin
```



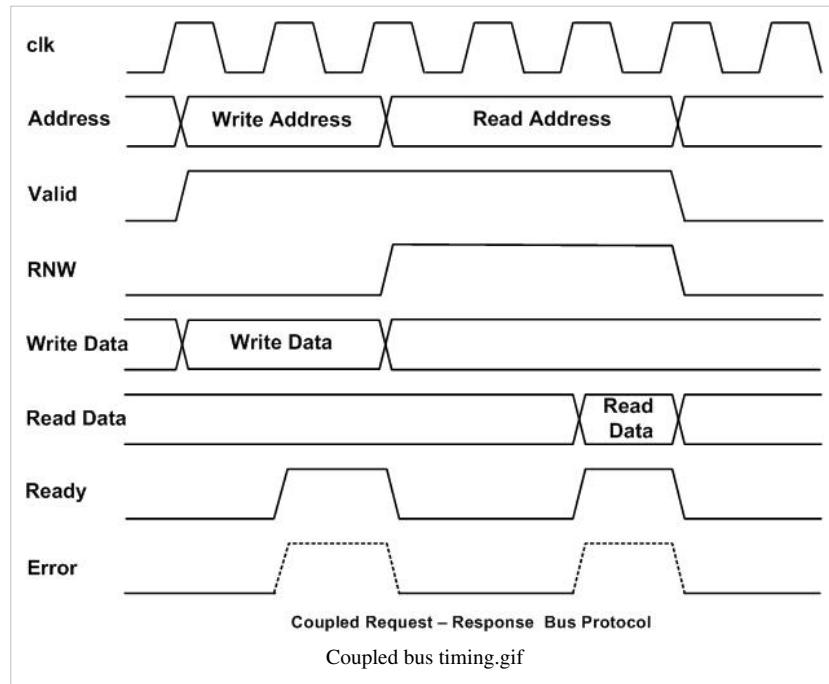
```

seq_item_port.get_next_item(req); // Start processing req item
repeat(req.delay) begin
  @(posedge BUS.clk);
end
BUS.valid <= 1;
BUS.addr <= req.addr;
BUS.rnw <= req.read_not_write;
if(req.read_not_write == 0) begin
  BUS.write_data <= req.write_data;
end
while(BUS.ready != 1) begin
  @(posedge BUS.clk);
end
// At end of the pin level bus transaction
// Copy response data into the req fields:
if(req.read_not_write == 1) begin
  req.read_data = BUS.read_data; // If read - copy returned read data
end
req.error = BUS.error; // Copy bus error status
BUS.valid <= 0; // End the pin level bus transaction
seq_item_port.item_done(); // End of req item
end
endtask: run_phase

endclass: bidirect_bus_driver

```

Note that the driver is sending back the response to the sequence by updating the fields within the req sequence\_item before making the item\_done() call. At the sequence end of the transaction, the sequence is blocked in the finish\_item() call until the item\_done() occurs, when it is unblocked, its req handle is still pointing to the req object which has had its response fields updated by the driver. This means that the sequence can reference the response contents of the req sequence\_item.



```

class bus_seq extends uvm_sequence #(bus_seq_item);

`uvm_object_utils(bus_seq)

bus_seq_item req;

rand int limit = 40; // Controls the number of iterations

function new(string name = "bus_seq");
  super.new(name);
endfunction

task body();
  req = bus_seq_item::type_id::create("req");

  repeat(limit) begin
    start_item(req);
    // The address is constrained to be within the address of the GPIO function
    // within the DUT, The result will be a request item for a read or a write
    assert(req.randomize() with {addr inside {[32'h0100_0000:32'h0100_001C]}});
    finish_item(req);
    // The req handle points to the object that the driver has updated with response data
    uvm_report_info("seq_body", req.convert2string());
  end
  endtask: body

endclass: bus_seq

```

( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

## Alternative Implementation Option

*Although this option is discussed below, the recommended way to implement this sequence driver use model is as in the preceding code.*

### Driver put, Sequence get\_response

Alternatively, the code in the driver could use a get() method to collect the request sequence\_item, this get() call would unblock the finish\_item() call in the sequence execution. However, the driver should use the put() method to signal back to the sequence that it has fully completed the bus transfer cycle, and the sequence should use a blocking call to the get\_response() method to wait for the driver to complete the transfer. Any response information from the pin level bus transaction can be sent from the driver to the sequence via the argument to the put() method.

```

// Alternative version of the driver run method
task run_phase( uvm_phase phase );
  bus_seq_item req;

```

```
bus_seq_item rsp;
// Default conditions:
BUS.valid <= 0;
BUS.rnw <= 1;
// Wait for reset to end
@(posedge BUS.resetn);
forever begin
  seq_item_port.get(req); // Start processing req item
  repeat(req.delay) begin
    @(posedge BUS.clk);
  end
  BUS.valid <= 1;
  BUS.addr <= req.addr;
  BUS.rnw <= req.read_not_write;
  if(req.read_not_write == 0) begin
    BUS.write_data <= req.write_data;
  end
  while(BUS.ready != 1) begin
    @(posedge BUS.clk);
  end
  // At end of the pin level bus transaction
  // Copy response data into the rsp fields:
  $cast(rsp, req.clone()); // Clone the req
  rsp.set_id_info(req); // Set the rsp id = req id
  if(rsp.read_not_write == 1) begin
    rsp.read_data = BUS.read_data; // If read - copy returned read data
  end
  rsp.error = BUS.error; // Copy bus error status
  BUS.valid <= 0; // End the pin level bus transaction
  seq_item_port.put(rsp); // put returns the response
end
endtask: run_phase

// Corresponding version of the sequence body method:
task body();
  bus_seq_item req;
  bus_seq_item rsp;

  req = bus_seq_item::type_id::create("req");

  repeat(limit) begin
    start_item(req);
    // The address is constrained to be within the address of the GPIO function
  end
endtask
```

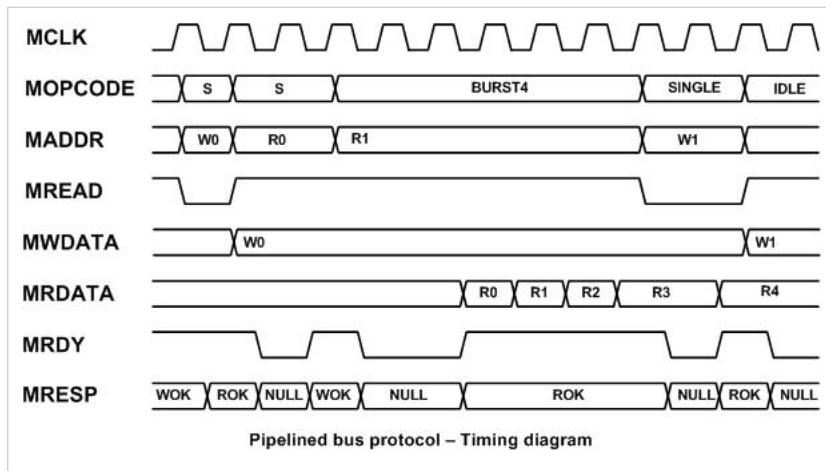
```
// within the DUT, The result will be a request item for a read or a write
assert(req.randomize() with {addr inside {[32'h0100_0000:32'h0100_001C]}});
finish_item(req);
get_response(rsp);
// The rsp handle points to the object that the driver has updated with response data
uvm_report_info("seq_body", rsp.convert2string());
end
endtask: body
```

For more information on this implementation approach, especially how to initialise the response item see the section on the get, put use model in the ***Driver/Sequence API*** article.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Driver/Pipelined

In a pipelined bus protocol a data transfer is broken down into two or more phases which are executed one after the other, often using different groups of signals on the bus. This type of protocol allows several transfers to be in progress at the same time with each transfer occupying one stage of the pipeline. The AMBA AHB bus is an example of a pipelined bus, it has two phases - the address phase and the data phase. During the address phase, the address and the bus control information, such as the opcode, is set up by the host, and then during the data phase the data transfer between the target and the host takes place. Whilst the data phase for one transfer is taking place on the second stage of the pipeline, the address phase for the next cycle can be taking place on the first stage of the pipeline. Other protocols such as OCP use more phases.



A pipelined protocol has the potential to increase the bandwidth of a system since, provided the pipeline is kept full, it increases the number of transfers that can take place over a given number of clock cycles. Using a pipeline also relaxes the timing requirements for target devices since it gives them extra time to decode and respond to a host access.

A pipelined protocol could be modelled with a simple bidirectional style, whereby the sequence sends a sequence item to the driver and the driver unblocks the sequence when it has completed the bus transaction. In reality, most I/O and register style accesses take place in this way. The drawback is that it lowers the bandwidth of the bus and does not stress test it. In order to implement a pipelined sequence-driver combination, there are a number of design considerations that need to be taken into account in order to support fully pipelined transfers:

- **Driver Implementation** - The driver needs to have multiple threads running, each thread needs to take a sequence item and take it through each of the pipeline stages.
- **Keeping the pipeline full** - The driver needs to unblock the sequencer to get the next sequence item so that the pipeline can be kept full
- **Sequence Implementation** - The sequence needs to have separate stimulus generation and response threads. The stimulus generation thread needs to continually send new bus transactions to the driver to keep the pipeline full.

## Recommended Implementation Pattern Using get and put

The most straight-forward way to model a pipelined protocol with a sequence and a driver is to use the `get()` and `put()` methods from the driver-sequencer API.

### Driver Implementation

In order to support pipelining, a driver needs to process multiple `sequence_items` concurrently. In order to achieve this, the drivers `run` method spawns a number of parallel threads each of which takes a `sequence_item` and executes it to completion on the bus. The number of threads required is equal to the number of stages in the pipeline. Each thread uses the `get()` method to acquire a new `sequence_item`, this unblocks the sequencer and the `finish_item()` method in the

sequence so that a new sequence item can be sent to the driver to fill the next stage of the pipeline.

In order to ensure that only one thread can call `get()` at a time, and also to ensure that only one thread attempts to drive the first phase of the bus cycle, a semaphore is used to lock access. The semaphore is grabbed at the start of the loop in the driver thread and is released at the end of the first phase, allowing another thread to grab the semaphore and take ownership.

At the end of the last phase in the bus cycle, the driver thread sends a response back to the sequence using the `put()` method. This returns the response to the originating sequence for processing.

In the code example a two stage pipeline is shown to illustrate the principles outlined.

```
//  
// This class implements a pipelined driver  
//  
class mbus_pipelined_driver extends uvm_driver #(mbus_seq_item);  
  
  `uvm_component_utils(mbus_pipelined_driver)  
  
  virtual mbus_if MBUS;  
  
  function new(string name = "mbus_pipelined_driver", uvm_component parent = null);  
    super.new(name, parent);  
  endfunction  
  
  // The two pipeline processes use a semaphore to ensure orderly execution  
  semaphore pipeline_lock = new(1);  
  //  
  // The run_phase(uvm_phase phase);  
  //  
  // This spawns two parallel transfer threads, only one of  
  // which can be active during the cmd phase, so implementing  
  // the pipeline  
  //  
  task run_phase(uvm_phase phase);  
  
    @(posedge MBUS.MRESETN);  
    @(posedge MBUS.MCLK);  
  
    fork  
      do_pipelined_transfer;  
      do_pipelined_transfer;  
    join  
  
  endtask  
  
  //
```

```
// This task is spawned in separate threads
//
task do_pipelined_transfer;
    mbus_seq_item req;

    forever begin
        // Unblocks when the semaphore is available:
        pipeline_lock.get();
        seq_item_port.get(req);
        accept_tr(req, $time);
        void'(begin_tr(req, "pipelined_driver"));
        MBUS.MADDR <= req.MADDR;
        MBUS.MREAD <= req.MREAD;
        MBUS.MOPCODE <= req.MOPCODE;
        @(posedge MBUS.MCLK);
        while(!MBUS.MRDY == 1) begin
            @(posedge MBUS.MCLK);
        end
        // End of command phase:
        // - unlock pipeline semaphore
        pipeline_lock.put();
        // Complete the data phase
        if(req.MREAD == 1) begin
            @(posedge MBUS.MCLK);
            while(MBUS.MRDY != 1) begin
                @(posedge MBUS.MCLK);
            end
            req.MRESP = MBUS.MRESP;
            req.MRDATA = MBUS.MRDATA;
        end
        else begin
            MBUS.MWDATA <= req.MWDATA;
            @(posedge MBUS.MCLK);
            while(MBUS.MRDY != 1) begin
                @(posedge MBUS.MCLK);
            end
            req.MRESP = MBUS.MRESP;
        end
        // Return the request as a response
        seq_item_port.put(req);
        end_tr(req);
    end
endtask: do_pipelined_transfer
```

```
endclass: mbus_pipelined_driver
```

## Sequence Implementation

### Unpipelined Accesses

Most of the time unpipelined transfers are required, since typical bus fabric is emulating what a software program does, which is to access single locations. For instance using the value read back from one location to determine what to do next in terms of reading or writing other locations.

In order to implement an unpipelined sequence that would work with the pipelined driver, the body() method would call start\_item(), finish\_item() and get\_response() methods in sequence. The get\_response() method blocks until the driver sends a response using its put() method at the end of the bus cycle. The following code example illustrates this:

```
//  
  
// This sequence shows how a series of unpipelined accesses to  
// the bus would work. The sequence waits for each item to finish  
// before starting the next.  
  
//  
  
class mbus_unpipelined_seq extends uvm_sequence #(mbus_seq_item);  
  
  `uvm_object_utils(mbus_unpipelined_seq)  
  
  logic[31:0] addr[10]; // To save addresses  
  logic[31:0] data[10]; // To save data for checking  
  
  int error_count;  
  
  function new(string name = "mbus_unpipelined_seq");  
    super.new(name);  
  endfunction  
  
  task body;  
  
    mbus_seq_item req = mbus_seq_item::type_id::create("req");  
  
    error_count = 0;  
    for(int i=0; i<10; i++) begin  
      start_item(req);  
      assert(req.randomize() with {MREAD == 0; MOPCODE == SINGLE; MADDR inside {[32'h0010_0000:32'h001F_FFFC]}});  
      addr[i] = req.MADDR;  
      data[i] = req.MWDATA;  
      finish_item(req);  
      get_response(req);  
    end  
  end
```

```

foreach (addr[i]) begin
  start_item(req);
  req.MADDR = addr[i];
  req.MREAD = 1;
  finish_item(req);
  get_response(req);
  if(data[i] != req.MRDATA) begin
    error_count++;
    `uvm_error("body", $sformatf("@%0h Expected data:%0h Actual data:%0h", addr[i], data[i], req.MRDATA))
  end
end
endtask: body

endclass: mbus_unpipelined_seq

```

**Note:** This example sequence has checking built-in, this is to demonstrate how a read data value can be used. The specific type of check would normally be done using a scoreboard.

### Pipelined Accesses

Pipelined accesses are primarily used to stress test the bus but they require a different approach in the sequence. A pipelined sequence needs to have a separate threads for generating the request sequence items and for handling the response sequence items.

The generation loop will block on each `finish_item()` call until one of the threads in the driver completes a `get()` call. Once the generation loop is unblocked it needs to generate a new item to have something for the next driver thread to `get()`. Note that a new request sequence item needs to be generated on each iteration of the loop, if only one request item handle is used then the driver will be attempting to execute its contents whilst the sequence is changing it.

In the example sequence, there is no response handling, the assumption is that checks on the data validity will be done by a scoreboard. However, with the `get()` and `put()` driver implementation, there is a response FIFO in the sequence which must be managed. In the example, the `response_handler` is enabled using the `use_response_handler()` method, and then the `response_handler` function is called everytime a response is available, keeping the sequences response FIFO empty. In this case the response handler keeps count of the number of transactions to ensure that the sequence only exist when the last transaction is complete.

```

//
// This is a pipelined version of the previous sequence with no blocking
// call to get_response();
// There is no attempt to check the data, this would be carried out
// by a scoreboard
//
class mbus_pipelined_seq extends uvm_sequence #(mbus_seq_item);

`uvm_object_utils(mbus_pipelined_seq)

logic[31:0] addr[10]; // To save addresses
int count; // To ensure that the sequence does not complete too early

```

```
function new(string name = "mbus_pipelined_seq");
    super.new(name);
endfunction

task body;

    mbus_seq_item req = mbus_seq_item::type_id::create("req");
    use_response_handler(1);
    count = 0;

    for(int i=0; i<10; i++) begin
        start_item(req);
        assert(req.randomize() with {MREAD == 0; MOPCODE == SINGLE; MADDR inside {[32'h0010_0000:32'h001F_FFFC]};});
        addr[i] = req.MADDR;
        finish_item(req);
    end

    foreach (addr[i]) begin
        start_item(req);
        req.MADDR = addr[i];
        req.MREAD = 1;
        finish_item(req);
    end

    // Do not end the sequence until the last req item is complete
    wait(count == 20);
endtask: body

// This response_handler function is enabled to keep the sequence response
// FIFO empty
function void response_handler(uvm_sequence_item response);
    count++;
endfunction: response_handler

endclass: mbus_pipelined_seq
```

If the sequence needs to handle responses, then the response handler function should be extended.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

## Alternative Implementation Pattern Using Events To Signal Completion

### Adding Completion Events to sequence\_items

In this implementation pattern, events are added to the sequence\_item to provide a means of signalling from the driver to the sequence that the driver has completed a specific phase. In the example, a uvm\_event\_pool is used for the events, and two methods are provided to trigger and to wait for events in the pool:

```
//  
// The mbus_seq_item is designed to be used with a pipelined bus driver.  
// It contains an event pool which is used to signal back to the  
// sequence when the driver has completed different pipeline stages  
//  
class mbus_seq_item extends uvm_sequence_item;  
  
// From the master to the slave  
rand logic[31:0] MADDR;  
rand logic[31:0] MWDATA;  
rand logic MREAD;  
rand mbus_opcode_e MOPCODE;  
  
// Driven by the slave to the master  
mbus_resp_e MRESP;  
logic[31:0] MRDATA;  
  
// Event pool:  
uvm_event_pool events;  
  
`uvm_object_utils(mbus_seq_item)  
  
function new(string name = "mbus_seq_item");  
    super.new(name);  
    events = get_event_pool();  
endfunction  
  
constraint addr_is_32 {MADDR[1:0] == 0;}  
  
// Wait for an event - called by sequence  
task wait_trigger(string evnt);  
    uvm_event e = events.get(evnt);  
    e.wait_trigger();  
endtask: wait_trigger  
  
// Trigger an event - called by driver  
task trigger(string evnt);
```

```
uvm_event e = events.get(evnt);
e.trigger();
endtask: trigger

// do_copy(), do_compare etc

endclass: mbus_seq_item
```

## Driver Signalling Completion using sequence\_item Events

The driver is almost identical to the get, put implementation except that it triggers the phase completed events in the sequence item rather than using a put() method signal to the sequence that a phase has completed and that there is response information available via the sequence\_item handle.

```
//

// This class implements a pipelined driver

class mbus_pipelined_driver extends uvm_driver #(mbus_seq_item);

`uvm_component_utils(mbus_pipelined_driver)

virtual mbus_if MBUS;

function new(string name = "mbus_pipelined_driver", uvm_component parent = null);
    super.new(name, parent);
endfunction

// the two pipeline processes use a semaphore to ensure orderly execution
semaphore pipeline_lock = new(1);

//

// The run_phase(uvm_phase phase);

//

// This spawns two parallel transfer threads, only one of
// which can be active during the cmd phase, so implementing
// the pipeline

//

task run_phase(uvm_phase phase);

    @(posedge MBUS.MRESETN);
    @(posedge MBUS.MCLK);

    fork
        do_pipelined_transfer;
        do_pipelined_transfer;
    join
```

```
endtask

//  
// This task has to be automatic because it is spawned  
// in separate threads  
//  
task do_pipelined_transfer;  
    mbus_seq_item req;  
  
    forever begin  
        pipeline_lock.get();  
        seq_item_port.get(req);  
        accept_tr(req, $time);  
        void'(begin_tr(req, "pipelined_driver"));  
        MBUS.MADDR <= req.MADDR;  
        MBUS.MREAD <= req.MREAD;  
        MBUS.MOPCODE <= req.MOPCODE;  
        @(posedge MBUS.MCLK);  
        while(!MBUS.MRDY == 1) begin  
            @(posedge MBUS.MCLK);  
        end  
        // End of command phase:  
        // - unlock pipeline semaphore  
        // - signal CMD_DONE  
        pipeline_lock.put();  
        req.trigger("CMD_DONE");  
        // Complete the data phase  
        if(req.MREAD == 1) begin  
            @(posedge MBUS.MCLK);  
            while(MBUS.MRDY != 1) begin  
                @(posedge MBUS.MCLK);  
            end  
            req.MRESP = MBUS.MRESP;  
            req.MRDATA = MBUS.MRDATA;  
        end  
        else begin  
            MBUS.MWDATA <= req.MWDATA;  
            @(posedge MBUS.MCLK);  
            while(MBUS.MRDY != 1) begin  
                @(posedge MBUS.MCLK);  
            end  
            req.MRESP = MBUS.MRESP;  
        end
```

```

    req.trigger("DATA_DONE");
    end_tr(req);
  end
endtask: do_pipelined_transfer

endclass: mbus_pipelined_driver

```

## Unpipelined Access Sequences

Unpipelined accesses are made from sequences which block, after completing the `finish_item()` call, by waiting for the data phase completed event. This enables code in the sequence body method to react to the data read back. An alternative way of implementing this type of sequence would be to overload the `finish_item` method so that it does not return until the data phase completed event occurs.

```

// Task: finish_item
//
// Calls super.finish_item but then also waits for the item's data phase
// event. This is notified by the driver when it has completely finished
// processing the item.
//
task finish_item( uvm_sequence_item item , int set_priority = -1 );
  // The "normal" finish_item()
  super.finish_item( item , set_priority );
  // Wait for the data phase to complete
  item.wait_trigger("DATA_DONE");
endtask

```

As in the previous example of an unpipelined sequence, the code example shown has a data integrity check, this is purely for illustrative purposes.

```

class mbus_unpipelined_seq extends uvm_sequence #(mbus_seq_item);

`uvm_object_utils(mbus_unpipelined_seq)

logic[31:0] addr[10]; // To save addresses
logic[31:0] data[10]; // To save addresses

int error_count;

function new(string name = "mbus_pipelined_seq");
  super.new(name);
endfunction

```

```

task body;

  mbus_seq_item req = mbus_seq_item::type_id::create("req");
  error_count = 0;

  for(int i=0; i<10; i++) begin
    start_item(req);
    assert(req.randomize() with {MREAD == 0; MOPCODE == SINGLE; MADDR inside {[32'h0010_0000:32'h001F_FFFC]};});
    addr[i] = req.MADDR;
    data[i] = req.MWDATA;
    finish_item(req);
    req.wait_trigger("DATA_DONE");
  end

  foreach(addr[i]) begin
    start_item(req);
    req.MADDR = addr[i];
    req.MREAD = 1;
    finish_item(req);
    req.wait_trigger("DATA_DONE");
    if(req.MRDATA != data[i]) begin
      error_count++;
      `uvm_error("body", $sformatf("@%0h Expected data:%0h Actual data:%0h", addr[i], data[i], req.MRDATA))
    end
  end
  endtask: body

endclass: mbus_unpipelined_seq

```

## Pipelined Access

The pipelined access sequence does not wait for the data phase completion event before generating the next sequence item. Unlike the get, put driver model, there is no need to manage the response FIFO, so in this respect this implementation model is more straight-forward.

```

class mbus_pipelined_seq extends uvm_sequence #(mbus_seq_item);

  `uvm_object_utils(mbus_pipelined_seq)

  logic[31:0] addr[10]; // To save addresses

  function new(string name = "mbus_pipelined_seq");
    super.new(name);
  endfunction

```

```
task body;

  mbus_seq_item req = mbus_seq_item::type_id::create("req");

  for(int i=0; i<10; i++) begin
    start_item(req);
    assert(req.randomize() with {MREAD == 0; MOPCODE == SINGLE; MADDR inside {[32'h0010_0000:32'h001F_FFFC]};});
    addr[i] = req.MADDR;
    finish_item(req);
  end

  foreach (addr[i]) begin
    start_item(req);
    req.MADDR = addr[i];
    req.MREAD = 1;
    finish_item(req);
  end
endtask: body

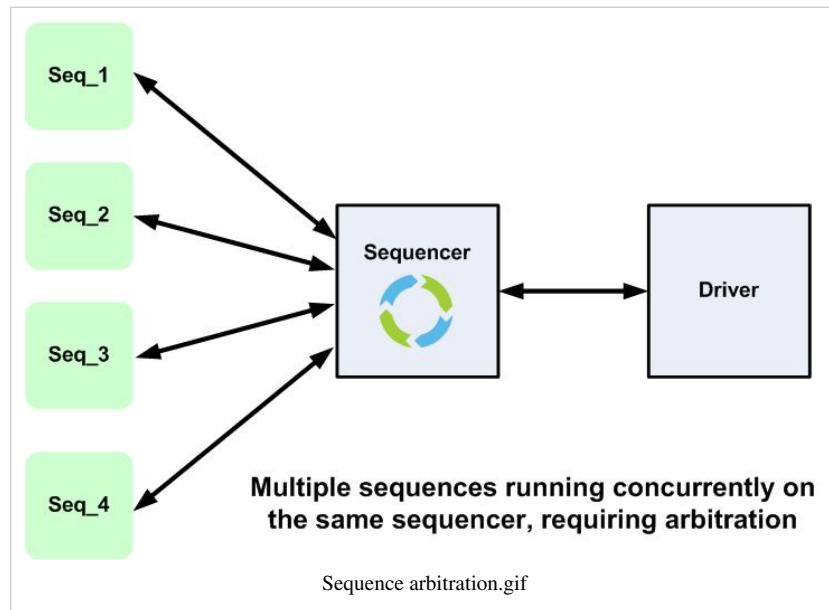
endclass: mbus_pipelined_seq
```

( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

# Sequences/Arbitration

The uvm\_sequencer has a built-in mechanism to arbitrate between sequences which could be running concurrently on a sequencer. The arbitration algorithm determines which sequence is granted access to send its sequence\_item to the driver. There is a choice of six arbitration algorithms which can be selected using the set\_arbitration() sequencer method from the controlling sequence.

Consider the example illustrated in the diagram. In this example we have four sequences which are running as sub-sequences of the main sequence. Each sequence sends sequence\_items to



the driver with its own id, the driver keeps count of how many of which ids have been received. The sequences have different priorities - seq\_1 and seq\_2 have the highest priority, then seq\_3, with seq\_4 having the lowest priority. The sequences generate sequence\_items at different time offsets, with seq\_1 starting first, followed by seq\_2 and so on.

The master sequence generation loop is shown in this code snippet:

```
task body;
  seq_1 = arb_seq::type_id::create("seq_1");
  seq_1.seq_no = 1;
  seq_2 = arb_seq::type_id::create("seq_2");
  seq_2.seq_no = 2;
  seq_3 = arb_seq::type_id::create("seq_3");
  seq_3.seq_no = 3;
  seq_4 = arb_seq::type_id::create("seq_4");
  seq_4.seq_no = 4;

  m_sequencer.set_arbitration(arb_type); // arb_type is set by the test
  fork
    begin
      repeat(4) begin
        #1; // Offset by 1
        seq_1.start(m_sequencer, this, 500); // Highest priority
      end
    end
    begin
      repeat(4) begin
        #1; // Offset by 1
        seq_2.start(m_sequencer, this, 500);
      end
    end
    begin
      repeat(4) begin
        #1; // Offset by 1
        seq_3.start(m_sequencer, this, 500);
      end
    end
    begin
      repeat(4) begin
        #1; // Offset by 1
        seq_4.start(m_sequencer, this, 500);
      end
    end
  join
endtask
```

```

    #2; // Offset by 2
    seq_2.start(m_sequencer, this, 500); // Highest priority
  end
end
begin
  repeat(4) begin
    #3; // Offset by 3
    seq_3.start(m_sequencer, this, 300); // Medium priority
  end
end
begin
  repeat(4) begin
    #4; // Offset by 4
    seq_4.start(m_sequencer, this, 200); // Lowest priority
  end
end
join

endtask: body

```

The six sequencer arbitration algorithms are best understood in the context of this example:

### **SEQ\_ARB\_FIFO (Default)**

This is the default sequencer arbitration algorithm. What it does is to send sequence\_items from the sequencer to the driver in the order they are received by the sequencer, regardless of any priority that has been set. In the example, this means that the driver receives sequence items in turn from seq\_1, seq\_2, seq\_3 and seq\_4. The resultant log file is as follows:

```

# UVM_INFO @ 0: uvm_test_top [Sequencer Arbitration selected:] SEQ_ARB_FIFO
# UVM_INFO @ 1: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:1
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 21: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 31: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:1 SEQ_4:0
# UVM_INFO @ 41: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:1 SEQ_4:1
# UVM_INFO @ 51: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:2
SEQ_3:1 SEQ_4:1
# UVM_INFO @ 61: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:1 SEQ_4:1
# UVM_INFO @ 71: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:2 SEQ_4:1

```

```

# UVM_INFO @ 81: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:2 SEQ_4:2
# UVM_INFO @ 91: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:3
SEQ_3:2 SEQ_4:2
# UVM_INFO @ 101: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:2 SEQ_4:2
# UVM_INFO @ 111: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:2
# UVM_INFO @ 121: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3
# UVM_INFO @ 131: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:3 SEQ_4:3
# UVM_INFO @ 141: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:4

```

## SEQ\_ARB\_WEIGHTED

With this algorithm, the sequence\_items to be sent to the driver are selected on a random basis but weighted with the sequence\_items from the highest priority sequence being sent first. In the example, this means that sequence\_items from seq\_1 and seq\_2 are selected on a random basis until all have been consumed, at which point the items from seq\_3 are selected, followed by seq\_4. The resultant log file illustrates this:

```

# UVM_INFO @ 0: uvm_test_top [Sequencer Arbitration selected:] SEQ_ARB_WEIGHTED
# UVM_INFO @ 1: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:1
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 21: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 31: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:2
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 41: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:3
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 51: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:3
SEQ_3:0 SEQ_4:1
# UVM_INFO @ 61: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:4
SEQ_3:0 SEQ_4:1
# UVM_INFO @ 71: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:4
SEQ_3:0 SEQ_4:1
# UVM_INFO @ 81: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:4
SEQ_3:1 SEQ_4:1
# UVM_INFO @ 91: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:4

```

```

SEQ_3:1 SEQ_4:2
# UVM_INFO @ 101: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:4
SEQ_3:2 SEQ_4:2
# UVM_INFO @ 111: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:2 SEQ_4:2
# UVM_INFO @ 121: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:3 SEQ_4:2
# UVM_INFO @ 131: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:2
# UVM_INFO @ 141: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:4

```

## SEQ\_ARB\_RANDOM

With this algorithm, the sequence\_items are selected on a random basis, irrespective of the priority level of their controlling sequences. The result in the example is that sequence\_items are sent to the driver in a random order irrespective of their arrival time at the sequencer and of the priority of the sequencer that sent them:

```

# UVM_INFO @ 0: uvm_test_top [Sequencer Arbitration selected:] SEQ_ARB_RANDOM
# UVM_INFO @ 1: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:1
# UVM_INFO @ 21: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:0
SEQ_3:0 SEQ_4:1
# UVM_INFO @ 31: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:0
SEQ_3:1 SEQ_4:1
# UVM_INFO @ 41: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:0
SEQ_3:1 SEQ_4:2
# UVM_INFO @ 51: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:1 SEQ_4:2
# UVM_INFO @ 61: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:1 SEQ_4:3
# UVM_INFO @ 71: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:2 SEQ_4:3
# UVM_INFO @ 81: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:1
SEQ_3:2 SEQ_4:3
# UVM_INFO @ 91: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:2 SEQ_4:3
# UVM_INFO @ 101: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:2
SEQ_3:2 SEQ_4:3
# UVM_INFO @ 111: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:2

```

```

SEQ_3:3 SEQ_4:3
# UVM_INFO @ 121: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3
# UVM_INFO @ 131: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:3 SEQ_4:3
# UVM_INFO @ 141: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:4

```

## SEQ\_ARB\_STRICT\_FIFO

The SEQ\_ARB\_STRICT\_FIFO algorithm sends sequence\_items to the driver based on their priority and their order in the FIFO, with highest priority items being sent in the order received. The result in the example is that the seq\_1 and seq\_2 sequence\_items are sent first interleaved with each other according to the order of their arrival in the sequencer queue, followed by seq\_3s sequence items, and then the sequence\_items for seq\_4.

```

# UVM_INFO @ 0: uvm_test_top [Sequencer Arbitration selected:] SEQ_ARB_STRICT_FIFO
# UVM_INFO @ 1: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:1
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 21: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 31: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:2
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 41: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 51: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:3
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 61: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 71: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 81: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:1 SEQ_4:0
# UVM_INFO @ 91: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:2 SEQ_4:0
# UVM_INFO @ 101: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:3 SEQ_4:0
# UVM_INFO @ 111: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:0
# UVM_INFO @ 121: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:1

```

```
# UVM_INFO @ 131: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:2
# UVM_INFO @ 141: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:4
```

## SEQ\_ARB\_STRICT\_RANDOM

This algorithm selects the sequence\_items to be sent in a random order but weighted by the priority of the sequences which are sending them. The effect in the example is that seq\_1 is selected randomly first and its sequence\_items are sent before the items from seq\_2, followed by seq\_3 and then seq\_4

```
# UVM_INFO @ 0: uvm_test_top [Sequencer Arbitration selected:] SEQ_ARB_STRICT_RANDOM
# UVM_INFO @ 1: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 21: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 31: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 41: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:1
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 51: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:2
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 61: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 71: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 81: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:1 SEQ_4:0
# UVM_INFO @ 91: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:2 SEQ_4:0
# UVM_INFO @ 101: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:3 SEQ_4:0
# UVM_INFO @ 111: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:0
# UVM_INFO @ 121: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:1
# UVM_INFO @ 131: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:2
# UVM_INFO @ 141: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:3
```

```
# UVM_INFO @ 151: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:4
```

## SEQ\_ARB\_USER

This algorithm allows a user defined method to be used for arbitration. In order to do this, the uvm\_sequencer must be extended to override the `user_priority_arbitration()` method. The method receives an argument which is the sequencer's queue of `sequence_items`, the user implemented algorithm needs to return an integer to select one of the `sequence_items` from the queue. The method is able to call on other methods implemented in the sequencer base class to establish the properties of each of the sequences in the queue. For instance, the priority of each sequence item can be established using the `get_seq_item_priority()` call as illustrated in the following example:

```
//
// Return the item with the mean average priority
//
function integer user_priority_arbitration(integer avail_sequences[$]);
    integer priority[] = new[avail_sequences.size]
    integer sum = 0;
    bit mean_found = 0;

    for (i = 0; i < avail_sequences.size(); i++) begin
        priority[i] = get_seq_item_priority(arb_sequence_q[avail_sequences[i]]);
        sum = sum + priority[i];
    end
    // take the mean priority
    sum = sum/avail_sequences.size();
    // Find the first sequence that matches this priority
    foreach(priority[i]) begin
        if(priority[i] == sum) begin
            return avail_sequences[i];
        end
    end
    // Otherwise return the mode average:
    return avail_sequences[(avail_sequences.size()/2];

endfunction: user_priority_arbitration
```

In the following example, the `user_priority_arbitration` method has been modified to always select the last `sequence_item` that was received, this is more or less the inverse of the default arbitration mechanism.

```
class seq_arb_sequencer extends uvm_sequencer #(seq_arb_item);

`uvm_component_utils(seq_arb_sequencer)

function new(string name = "seq_arb_sequencer", uvm_component parent = null);
```

```

super.new(name, parent);
endfunction

// This method overrides the default user method
// It returns the last item in the sequence queue
function integer user_priority_arbitration(integer avail_sequences[$]);
  int end_index;
  end_index = avail_sequences.size() - 1;
  return (avail_sequences[end_index]);
endfunction // user_priority_arbitration

endclass: seq_arb_sequencer

```

In the example, using this algorithm has the effect of sending the sequence\_items from seq\_4, followed by seq\_3, seq\_2 and then seq\_1.

```

# UVM_INFO @ 0: uvm_test_top [Sequencer Arbitration selected:] SEQ_ARB_USER
# UVM_INFO @ 1: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:1
# UVM_INFO @ 21: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:2
# UVM_INFO @ 31: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:3
# UVM_INFO @ 41: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:4
# UVM_INFO @ 51: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:1 SEQ_4:4
# UVM_INFO @ 61: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:2 SEQ_4:4
# UVM_INFO @ 71: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:3 SEQ_4:4
# UVM_INFO @ 81: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:4 SEQ_4:4
# UVM_INFO @ 91: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:0
SEQ_3:4 SEQ_4:4
# UVM_INFO @ 101: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:0
SEQ_3:4 SEQ_4:4
# UVM_INFO @ 111: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:0
SEQ_3:4 SEQ_4:4
# UVM_INFO @ 121: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:1
SEQ_3:4 SEQ_4:4
# UVM_INFO @ 131: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:2
SEQ_3:4 SEQ_4:4

```

```
# UVM_INFO @ 141: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:4 SEQ_4:4
# UVM_INFO @ 151: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:4 SEQ_4:4
```

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# Sequences/Priority

The UVM sequence use model allows multiple sequences to access a driver concurrently. The sequencer contains an **arbitration mechanism** that determines when a sequence\_item from a sequence will be sent to a driver. When a sequence is started using the start() method one of the arguments that can be passed is an integer indicating the priority of that sequence. The higher the value of the integer, the higher the priority of the sequence. This priority can be used with the SEQ\_ARB\_WEIGHTED, SEQ\_ARB\_STRICT\_FIFO, and SEQ\_ARB\_STRICT\_RANDOM arbitration mechanisms, (and possibly the SEQ\_ARB\_USER algorithm, if it handles priority) to ensure that a sequence has the desired priority.

**Note:** the remaining sequencer arbitration mechanisms do not take the sequence priority into account.

For instance, if a bus fabric master port has 3 sequences running on it to model software execution accesses (high priority), video data block transfer (medium priority) and irritant data transfer (low priority) then the hierarchical sequence controlling them would look like:

```
// Coding tip: Create an enumerated type to represent the different priority levels
typedef enum {HIGH_PRIORITY = 500, MED_PRIORITY = 200, LOW_PRIORITY = 50} seq_priority_e;

// In the hierarchical sequence:

task body();
    op_codes = cpu_sw_seq::type_id::create("op_codes");
    video = video_data_seq::type_id::create("video");
    irritant = random_access_seq::type_id::create("irritant");

    m_sequencer.set_arbitration(SEQ_ARB_STRICT_FIFO); // Arbitration mechanism that uses priority
    fork
        op_codes.start(m_sequencer, this, HIGH_PRIORITY);
        video.start(m_sequencer, this, MED_PRIORITY);
        irritant.start(m_sequencer, this, LOW_PRIORITY);
    join
endtask: body
```

When it is necessary to override the sequencer priority mechanism to model an interrupt or a high priority DMA transfer, then the sequencer **locking** mechanism should be used.

# Sequences/LockGrab

---

There are a number of modelling scenarios where one sequence needs to have exclusive access to a driver via a sequencer. One example of this type of scenario is a sequence which is responding to an interrupt. In order to accomodate this requirement, the uvm\_sequencer has a locking mechanism which is implemented using two calls - lock() and grab(). In terms of modelling, a lock might be used to model a prioritised interrupt and a grab might be used to model a non-maskable interrupt (NMI). The lock() and grab() calls have antidote calls to release a lock and these are unlock() and ungrab().

## lock

The sequencer lock method is called from a sequence and its effect is that the calling sequence will be granted exclusive access to the driver when it gets its next slot via the sequencer arbitration mechanism. Once lock is granted, no other sequences will be able to access the driver until the sequence issues an unlock() call which will then release the lock. The method is blocking and does not return until lock has been granted.

## grab

The grab method is similar to the lock method, except that it takes immediate effect and will grab the next sequencer arbitration slot, overriding any sequence priorities in place. The only thing that stops a sequence from grabbing a sequencer is a pre-existing lock() or grab() condition.

## unlock

The unlock sequencer function is called from within a sequence to give up its lock or grab. A locking sequence must call unlock before completion, otherwise the sequencer will remain locked.

## ungrab

The ungrab function is an alias of unlock.

## Related functions:

### is\_blocked

A sequence can determine if it is blocked by a sequencer lock condition by making this call. If it returns a 0, then the sequence is not blocked and will get a slot in the sequencer arbitration. However, the sequencer may get locked before the sequence has a chance to make a start\_item() call.

**is\_grabbed**

If a sequencer returns a 1 from this call, then it means that it has an active lock or grab in progress.

**current\_grabber**

This function returns the handle of the sequence which is currently locking the sequencer. This handle could be used to stop the locking sequence or to call a function inside it to unlock it.

**Gotchas:**

When a hierarchical sequence locks a sequencer, then its child sequences will have access to the sequencer. If one of the child sequences issues a lock, then the parent sequence will not be able to start any parallel sequences or send any sequence\_items until the child sequence has unlocked.

A locking or grabbing sequence must always unlock before it completes, otherwise the sequencer will become deadlocked.

**Example:**

The following example has 4 sequences which run in parallel threads. One of the threads has a sequence that does a lock, and a sequence that does a grab, both are functionally equivalent with the exception of the lock or grab calls.

Locking sequence:

```
class lock_seq extends uvm_sequence #(seq_arb_item);  
  
  `uvm_object_utils(lock_seq)  
  
  int seq_no;  
  
  function new(string name = "lock_seq");  
    super.new(name);  
  endfunction  
  
  task body();  
    seq_arb_item REQ;  
  
    if(m_sequencer.is_blocked(this)) begin  
      uvm_report_info("lock_seq", "This sequence is blocked by an existing lock");  
    end else begin  
      uvm_report_info("lock_seq", "This sequence is not blocked by an existing lock");  
    end  
  
    // Lock call - which blocks until it is granted  
    m_sequencer.lock(this);  
  
    if(m_sequencer.is_grabbed()) begin
```

```

if(m_sequencer.current_grabber() != this) begin
  uvm_report_info("lock_seq", "Lock sequence waiting for current grab or lock to complete");
end
end

REQ = seq_arb_item::type_id::create("REQ");
REQ.seq_no = 6;
repeat(4) begin
  start_item(REQ);
  finish_item(REQ);
end
// Unlock call - must be issued
m_sequencer.unlock(this);
endtask: body

endclass: lock_seq

```

Grabbing sequence:

```

class grab_seq extends uvm_sequence #(seq_arb_item);
`uvm_object_utils(grab_seq)

function new(string name = "grab_seq");
  super.new(name);
endfunction

task body();
  seq_arb_item REQ;

  if(m_sequencer.is_blocked(this)) begin
    uvm_report_info("grab_seq", "This sequence is blocked by an existing lock");
  end else begin
    uvm_report_info("grab_seq", "This sequence is not blocked by an existing lock");
  end

  // Grab call which blocks until grab has been granted
  m_sequencer.grab(this);

  if(m_sequencer.is_grabbed()) begin
    if(m_sequencer.current_grabber() != this) begin
      uvm_report_info("grab_seq", "Grab sequence waiting for current grab or lock to complete");
    end
  end
end

```

```

REQ = seq_arb_item::type_id::create("REQ");
REQ.seq_no = 5;
repeat(4) begin
  start_item(REQ);
  finish_item(REQ);
end
// Ungrab which must be called to release the grab (lock)
m_sequencer.ungrab(this);
endtask: body

endclass: grab_seq

```

The overall controlling sequence runs four sequences which send sequence\_items to the driver with different levels of priority. The driver reports from which sequence it has received a sequence\_item. The first grab\_seq in the fourth thread jumps the arbitration queue. The lock\_seq takes its turn and blocks the second grab\_seq, which then executes immediately the lock\_seq completes.

```

task body();
  seq_1 = arb_seq::type_id::create("seq_1");
  seq_1.seq_no = 1;
  seq_2 = arb_seq::type_id::create("seq_2");
  seq_2.seq_no = 2;
  seq_3 = arb_seq::type_id::create("seq_3");
  seq_3.seq_no = 3;
  seq_4 = arb_seq::type_id::create("seq_4");
  seq_4.seq_no = 4;
  grab = grab_seq::type_id::create("grab");
  lock = lock_seq::type_id::create("lock");

  m_sequencer.set_arbitration(arb_type);
  fork begin // Thread 1
    repeat(10) begin
      #1;
      seq_1.start(m_sequencer, this, 500); // Highest priority
    end
  end begin // Thread 2
    repeat(10) begin
      #2;
      seq_2.start(m_sequencer, this, 500); // Highest priority
    end
  end begin // Thread 3
    repeat(10) begin
      #3;
      seq_3.start(m_sequencer, this, 300); // Medium priority
    end
  end
end

```

```

    end
end begin // Thread 4
fork
repeat(2) begin
#4;
seq_4.start(m_sequencer, this, 200); // Lowest priority
end
#10 grab.start(m_sequencer, this, 50);
join
repeat(1) begin
#4 seq_4.start(m_sequencer, this, 200);
end
fork
lock.start(m_sequencer, this, 200);
#20 grab.start(m_sequencer, this, 50);
join
end join

endtask: body

```

The resultant simulation transcript is as follows:

```

UVM_INFO @ 0: reporter [RNTST] Running test arb_test...
# UVM_INFO @ 0: uvm_test_top [Sequencer Arbitration selected:] SEQ_ARB_FIFO
# UVM_INFO @ 1: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0 GRAB:0 LOCK:0
# UVM_INFO @ 10: reporter [grab_seq] This sequence is blocked by an existing lock in
place
# UVM_INFO @ 11: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0 GRAB:1 LOCK:0
# UVM_INFO @ 21: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0 GRAB:2 LOCK:0
# UVM_INFO @ 31: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0 GRAB:3 LOCK:0
# UVM_INFO @ 41: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:0
SEQ_3:0 SEQ_4:0 GRAB:4 LOCK:0
# UVM_INFO @ 51: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:1 SEQ_2:1
SEQ_3:0 SEQ_4:0 GRAB:4 LOCK:0
# UVM_INFO @ 61: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:0 SEQ_4:0 GRAB:4 LOCK:0
# UVM_INFO @ 71: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:1 SEQ_4:0 GRAB:4 LOCK:0
# UVM_INFO @ 81: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:1
SEQ_3:1 SEQ_4:1 GRAB:4 LOCK:0
# UVM_INFO @ 91: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:2 SEQ_2:2

```

```
SEQ_3:1 SEQ_4:1 GRAB:4 LOCK:0
# UVM_INFO @ 101: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:1 SEQ_4:1 GRAB:4 LOCK:0
# UVM_INFO @ 111: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:2 SEQ_4:1 GRAB:4 LOCK:0
# UVM_INFO @ 121: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:2
SEQ_3:2 SEQ_4:2 GRAB:4 LOCK:0
# UVM_INFO @ 131: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:3 SEQ_2:3
SEQ_3:2 SEQ_4:2 GRAB:4 LOCK:0
# UVM_INFO @ 141: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:2 SEQ_4:2 GRAB:4 LOCK:0
# UVM_INFO @ 151: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:2 GRAB:4 LOCK:0
# UVM_INFO @ 161: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:4 LOCK:0
# UVM_INFO @ 161: reporter [lock_seq] This sequence is not blocked by an existing lock
# UVM_INFO @ 171: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:4 LOCK:1
# UVM_INFO @ 181: reporter [grab_seq] This sequence is not blocked by an existing lock in
place
# UVM_INFO @ 181: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:4 LOCK:2
# UVM_INFO @ 191: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:4 LOCK:3
# UVM_INFO @ 201: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:4 LOCK:4
# UVM_INFO @ 211: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:5 LOCK:4
# UVM_INFO @ 221: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:6 LOCK:4
# UVM_INFO @ 231: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:7 LOCK:4
# UVM_INFO @ 241: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:3
SEQ_3:3 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 251: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:4 SEQ_2:4
SEQ_3:3 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 261: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:5 SEQ_2:4
SEQ_3:3 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 271: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:5 SEQ_2:4
SEQ_3:4 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 281: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:5 SEQ_2:5
SEQ_3:4 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 291: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:6 SEQ_2:5
SEQ_3:4 SEQ_4:3 GRAB:8 LOCK:4
```

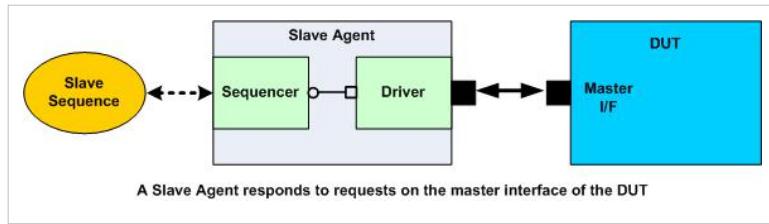
```
# UVM_INFO @ 301: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:6 SEQ_2:5
SEQ_3:5 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 311: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:6 SEQ_2:6
SEQ_3:5 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 321: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:7 SEQ_2:6
SEQ_3:5 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 331: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:7 SEQ_2:6
SEQ_3:6 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 341: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:7 SEQ_2:7
SEQ_3:6 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 351: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:8 SEQ_2:7
SEQ_3:6 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 361: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:8 SEQ_2:7
SEQ_3:7 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 371: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:8 SEQ_2:8
SEQ_3:7 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 381: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:9 SEQ_2:8
SEQ_3:7 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 391: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:9 SEQ_2:8
SEQ_3:8 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 401: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:9 SEQ_2:9
SEQ_3:8 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 411: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:10 SEQ_2:9
SEQ_3:8 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 421: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:10 SEQ_2:9
SEQ_3:9 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 431: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:10 SEQ_2:10
SEQ_3:9 SEQ_4:3 GRAB:8 LOCK:4
# UVM_INFO @ 441: uvm_test_top.m_driver [RECEIVED_SEQ] Access totals: SEQ_1:10 SEQ_2:10
SEQ_3:10 SEQ_4:3 GRAB:8 LOCK:4
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm> ).**

# Sequences/Slave

## Overview

A slave sequence is used with a driver that responds to events on an interface rather than initiating them. This type of functionality is usually referred to as a responder.



A responder can be implemented in several ways, for instance a simple bus oriented responder could be implemented as a uvm\_component interacting with a slave interface and reading and writing from memory according to the requests from the bus master.

The advantage of using a slave sequence is that the way in which the slave responds can be easily changed. One interesting characteristic of responder functionality is that it is not usually possible to predict when a response to a request will be required. For this reason slave sequences tend to be implemented as long-lasting sequences, i.e. they last for the whole of the simulation providing the responder functionality.

In this article, two approaches to implementing a slave sequence are described:

- Using a single sequence item
- Using a sequence item for each slave phase (in the APB example used as an illustration there are two phases).

In both cases, the sequence and the driver loop through the following transitions:

1. Slave sequence sends a request to the driver - "Tell me what to do"
2. Driver detects a bus level request and returns the information back to the sequence - "This is what you should do"
3. Slave sequence does what it needs to do to prepare a response and then sends a response item to the driver - "Here you go"
4. Driver completes the bus level response with the contents of the response item, completes handshake back to the sequence - "Thank you"

## Single Sequence Item Implementation

In this implementation, only one sequence item is used for both the request and response halves of the slave loop.

### Sequence item

The response properties of a slave sequence item should be marked as rand and the properties driven during the master request should not. If you were to compare a master sequence\_item and a slave sequence\_item for the same bus protocol, then you would find that which properties were marked rand and which were not would be reversed.

```
class apb_slave_seq_item extends uvm_sequence_item;
  `uvm_object_utils(apb_slave_seq_item)

  //-----
  // Data Members (Outputs rand, inputs non-rand)
  //-----
  logic[31:0] addr;
  logic[31:0] wdata;
```

```

logic rw;

rand logic[31:0] rdata;
rand logic slv_err;
rand int delay;

//-----
// Constraints
//-----
constraint delay_bounds {
    delay inside {[0:2]};
}

constraint error_dist {
    slv_err dist {0 := 80, 1 := 20};
}

//-----
// Methods
//-----
extern function new(string name = "apb_slave_seq_item");
extern function void do_copy(uvm_object rhs);
extern function bit do_compare(uvm_object rhs, uvm_comparer comparer);
extern function string convert2string();
extern function void do_print(uvm_printer printer);
extern function void do_record(uvm_recorder recorder);

endclass:apb_slave_seq_item

```

## Sequence and Driver

A slave device receives a request from the master and responds back with the relevant information. In order for the slave sequence to model this behavior, it needs to use multiple objects of the slave sequence item to accomplish the different phases. Depending on the bus protocol, at least two objects of the slave sequence item need to be used; one to get the request information and the second to return the response.

The process starts by the sequence sending a request object to the driver. The sequence does not need to initialize the contents of this object since it will be populated by the driver when it identifies a valid request on the pin level interface. When the driver calls the item\_done() method, the slave sequence is unblocked and can use the contents of the request object to prepare the response object.

Once the slave sequence has processed the request, it creates and populates a response object and calls start/finish\_item() to send it to the driver. The driver uses the response data to complete the response part of the pin level transfer and then unblocks the slave sequence execution by calling item\_done(). The slave sequence may randomize the response data in some way.

Example slave sequence code:

```

task apb_slave_sequence::body;
  apb_slave_agent_config m_cfg = apb_slave_agent_config::get_config(m_sequencer);
  apb_slave_seq_item req;
  apb_slave_seq_item rsp;

  wait (m_cfg.APB.PRESETn);
  forever begin

    req = apb_slave_seq_item::type_id::create("req");
    rsp = apb_slave_seq_item::type_id::create("rsp");

    // Slave request:
    start_item(req);
    finish_item(req);

    // Slave response:
    if (req.rw) begin
      memory[req.addr] = req.wdata;
    end
    start_item(rsp);
    rsp.copy(req);
    assert (rsp.randomize() with {if(!rsp.rw) rsp.rdata == memory[rsp.addr];});
    finish_item(rsp);
  end

  endtask:body

```

Example slave driver code:

```

task apb_slave_driver::run_phase(uvm_phase phase);
  apb_slave_seq_item req;
  apb_slave_seq_item rsp;

  forever
    begin
      if (!APB.PRESETn) begin
        APB.PREADY = 1'b0;
        APB.PSLVERR = 1'b0;
        @(posedge APB.PCLK);
      end
      else begin
        // Setup Phase
        seq_item_port.get_next_item(req);

        // Setup phase activity
      end
    end

```

```

    seq_item_port.item_done();

    // Access Phase
    seq_item_port.get_next_item(rsp);

    // Access phase activity

    seq_item_port.item_done();
end
end

endtask: run_phase

```

Example	Download Link
Complete APB3 Slave Agent	( <a href="#">download source code examples online at [1]</a> ).

## Multiple Sequence Items Implementation

### Sequence items

In this implementation, we will use more than one sequence item (usually called phase level sequence items) to implement the slave functionality. Depending on the bus protocol, at least two sequence items will be required; one to implement the request phase and a second to implement the response phase. One way of looking at this is to consider it as the single sequence implementation sliced in two. However, with more complex protocols there could be more than two phase level sequence items.

The request sequence item will contain those data members that are not random and will, consequently, have no constraints. The response sequence item will contain all those random data members in addition to some data members that overlap with the request sequence item. Those overlapping members are needed by the response sequence item to make some decisions, for example a read/write bit is required by the driver to know if it needs to drive the read data bus with valid data.

For example, this is the APB3 slave setup (request) sequence item.

```

class apb_slave_setup_item extends apb_sequence_item;
  `uvm_object_utils(apb_slave_setup_item)

  //-----
  // Data Members (Outputs rand, inputs non-rand)
  //-----
  logic[31:0] addr;
  logic[31:0] wdata;
  logic rw;

endclass

```

And this is the access (response) sequence item. Note the rand data members and the constraints.

```
class apb_slave_access_item extends apb_sequence_item;
  `uvm_object_utils(apb_slave_setup_item)

  //-----
  // Data Members (Outputs rand, inputs non-rand)
  //-----
  rand logic rw;

  rand logic[31:0] rdata;
  rand logic slv_err;
  rand int delay;

  constraint delay_bounds {
    delay inside {[0:2]};
  }

  constraint error_dist {
    slv_err dist {0 := 80, 1 := 20};
  }
endclass
```

## Sequence and Driver

The slave sequence and driver in this implementation are very similar to that described above. However, the major difference is that the sequencer and the driver are parameterised with the apb\_sequence\_item base class so that both request and response sequence item type objects can be passed between the sequence and the driver. The driver casts the received sequence item object to the appropriate request or response sequence item type before it accesses the contents.

For the sequence, the only difference will be in the parameterization of the class template and no casting is required at all. The sequence body remains the same.

```
class apb_slave_sequence extends uvm_sequence #(apb_sequence_item);
```

As a consequence, the Sequencer's class definition will change as well.

```
class apb_slave_sequencer extends uvm_sequencer #(apb_sequence_item);
```

The driver's class definition will change too.

```
class apb_slave_driver extends uvm_driver #(apb_sequence_item, apb_sequence_item);
```

The run\_phase of the driver will always use the uvm\_sequence\_item to get\_next\_item and then cast the received sequence item to the appropriate/correct type.

```
task apb_slave_driver::run_phase(uvm_phase phase);
  apb_sequence_item item;
  apb_slave_setup_item req;
  apb_slave_access_item rsp;
```

```
forever
begin
  if (!APB.PRESETn) begin
    ...
  end
  else begin
    seq_item_port.get_next_item(item);
    if ($cast(req, item))
      begin
        ...
      end
    else
      `uvm_error("CASTFAIL", "The received sequence item is not a request seq_item");
    seq_item_port.item_done();
  end
  else
    `uvm_error("CASTFAIL", "The received sequence item is not a response seq_item");
  seq_item_port.item_done();
end
end
endtask: run_phase
```

Example	Download Link
Complete APB3 Slave Agent using multiple sequence items	( <a href="#">download source code examples online at [1]</a> ).

# Stimulus/Signal Wait

In the general case, synchronising to hardware events is taken care of in UVM testbenches by drivers and monitors. However, there are some cases where it is useful for a sequence or a component to synchronise to a hardware event such as a clock without interacting with a driver or a monitor. This can be facilitated by adding methods to an object containing a virtual interface (usually a configuration object) that block until a hardware event occurs on the virtual interface. A further refinement is to add a `get_config()` method which allows a component to retrieve a pointer to its configuration object based on its scope within the UVM testbench component hierarchy.

## Additions to the configuration object

In order to support the use of a `wait_for_interface_signal` method, hardware synchronisation methods have to be added to a configuration object. These hardware synchronisation methods reference signals within the virtual interface(s) which the configuration object contains handles for. Examples of hardware synchronisation methods include:

- `wait_for_clock()`
- `wait_for_reset()`
- `wait_for_interrupt()`
- `interrupt_cleared()`

An example:

```
class bus_agent_config extends uvm_object;

`uvm_object_utils(bus_agent_config)

virtual bus_if BUS; // This is the virtual interface with the signals to wait on

function new(string name = "bus_agent_config");
    super.new(name);
endfunction

// 
// Task: wait_for_reset
// 
// This method waits for the end of reset.
task wait_for_reset;
    @( posedge BUS.resetn );
endtask

// 
// Task: wait_for_clock
// 
// This method waits for n clock cycles.
task wait_for_clock( int n = 1 );
    repeat( n ) begin
```

```

@( posedge BUS.clk );
end
endtask

// 
// Task: wait_for_error
//
task wait_for_error;
@(posedge error);
endtask: wait_for_error

//
// Task: wait_for_no_error
//
task wait_for_no_error;
@(negedge error);
endtask: wait_for_no_error

endclass: bus_agent_config

```

## Using the wait\_for\_interface\_signal method

In order to use the `wait_for_xxx()` method, the sequence or component must first ensure that it has a valid pointer to the configuration object. The pointer may already have been set up during construction or it may require the sequence or component to call the `get_config()` static method. Since sequences are not part of the UVM component hierarchy, they need to reference it via the `m_sequencer` pointer.

Once the local configuration object pointer is valid, the method can be called via the configuration object handle.

A sequence example:

```

class bus_seq extends uvm_sequence #(bus_seq_item);
`uvm_object_utils(bus_seq)

bus_seq_item req;
bus_agent_config m_cfg;

rand int limit = 40; // Controls the number of iterations

function new(string name = "bus_seq");
super.new(name);
endfunction

task body;
int i = 5;
req = bus_seq_item::type_id::create("req");

```

```

// Get the configuration object
if(!uvm_config_db #(bus_agent_config)::get(null, get_full_name(), "config", m_cfg)) begin
  `uvm_error("BODY", "bus_agent_config not found")
end

repeat(limit)
begin
  start_item(req);
  // The address is constrained to be within the address of the GPIO function
  // within the DUT, The result will be a request item for a read or a write
  if(!req.randomize() with {addr inside {[32'h0100_0000:32'h0100_001C]};}) begin
    `uvm_error("body", "randomization failed for req")
  end
  finish_item(req);
  // Wait for interface clocks:
  m_cfg.wait_for_clock(i);
  i++;
  // The req handle points to the object that the driver has updated with response data
  uvm_report_info("seq_body", req.convert2string());
end
endtask: body

endclass: bus_seq

```

A component example:

```

// 
// A coverage monitor that should ignore coverage collected during an error condition:
// 

class transfer_link_coverage_monitor extends uvm_subscriber #(trans_link_item);

`uvm_component_utils(transfer_link_coverage_monitor)

T pkt;

// Error monitor bit
bit no_error;
// Configuration:
bus_agent_config m_cfg;

covergroup tlcm_1;
  HDR: coverpoint pkt.hdr;
  SPREAD: coverpoint pkt.payload {

```

```
bins spread[] {[0:1023], [1024:8095], [8096:$]}
```

```
}
```

```
cross HDR, SPREAD;
```

```
endgroup: tlcm_1
```

```
function new(string name = "transfer_link_coverage_monitor", uvm_component_parent = null);
```

```
super.new(name, parent);
```

```
tlcm_1 = new;
```

```
endfunction
```

```
// The purpose of the run method is to monitor the state of the error
```

```
// line
```

```
task run_phase( uvm_phase phase );
```

```
no_error = 0;
```

```
// Get the configuration
```

```
if(!uvm_config_db #(bus_agent_config)::get(this, "", "bus_agent_config", m_cfg)) begin
```

```
 `uvm_error("run_phase", "Configuration error: unable to find bus_agent_config")
```

```
end
```

```
m_cfg.wait_for_reset; // Nothing valid until reset is over
```

```
no_error = 1;
```

```
forever begin
```

```
m_cfg.wait_for_error; // Blocks until an error occurs
```

```
no_error = 0;
```

```
m_cfg.wait_for_no_error; // Blocks until the error is removed
```

```
end
```

```
endtask: run_phase
```

```
function write(T t);
```

```
pkt = t;
```

```
if(no_error == 1) begin // Don't sample if there is an error
```

```
tlcm_1.sample();
```

```
end
```

```
endfunction: write
```

```
endclass: transfer_link_coverage_monitor
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Stimulus/Interrupts

In hardware terms, an interrupt is an event which triggers a new thread of processing. This new thread can either take the place of the current execution thread, which is the case with an interrupt service routine or it can be used to wake up a sleeping process to initiate hardware activity. Either way, the interrupt is treated as a sideband signal or event which is not part of the main bus or control protocol.

In CPU based systems, interrupts are typically managed in hardware by interrupt controllers which can be configured to accept multiple interrupt request lines, to enable and disable interrupts, prioritise them and latch their current status. This means that a typical CPU only has one interrupt request line which comes from an interrupt controller and when the CPU responds to the interrupt it accesses the interrupt controller to determine the source and takes the appropriate action to clear it down. Typically, the role of the testbench is to verify the hardware implementation of the interrupt controller, however, there are circumstances where interrupt controller functionality has to be implemented in the testbench.

In some systems, an interrupt service route is re-entrant, meaning that if an ISR is in progress and a higher priority interrupt occurs, then the new interrupt triggers the execution of a new ISR thread which returns control to the first ISR on completion.

A stimulus generation flow can be adapted to take account of hardware interrupts in one of several ways:

- Exclusive execution of an ISR sequence
- Prioritised execution of an ISR sequence or sequences
- Hardware triggered sequences

## Hardware Interrupt Detection

Each one of these approaches relies on an interrupt being detected via a transition on a virtual interface signal. The most convenient way to detect this signal transition is to add a *wait\_for\_hardware\_event* task into the configuration object that contains a pointer to the virtual interface. The overall control sequence can then call a *wait\_for\_event* task as a sideband activity rather than use an agent to implement the interrupt interface.

## Exclusive ISR Sequence

The simplest way to model interrupt handling is to trigger the execution of a sequence that uses the *grab()* method to get exclusive access to the target sequencer. This is a disruptive way to interrupt other stimulus generation that is taking place, but it does emulate what happens when an ISR is triggered on a CPU. The interrupt sequence cannot be interrupted itself, and must make an *ungrab()* call before it completes.

The interrupt monitor is usually implemented in a forked process running in a control or virtual sequence. The forked process waits for an interrupt, then starts the ISR sequence. When the ISR sequence ends, the loop starts again by waiting for the next interrupt.

```
//
// Sequence runs a bus intensive sequence on one thread
// which is interrupted by one of four interrupts
//
class int_test_seq extends uvm_sequence #(bus_seq_item);
`uvm_object_utils(int_test_seq)
```

```

function new (string name = "int_test_seq");
  super.new(name);
endfunction

task body;
  set_ints setup_ints; // Main activity on the bus interface
  ISR ISR; // Interrupt service routine
  int_config i_cfg; // Config containing wait_for_IRQx tasks

  setup_ints = set_ints::type_id::create("setup_ints");
  ISR = ISR::type_id::create("ISR");
  i_cfg = int_config::get_config(m_sequencer); // Get the config

  // Forked process - two levels of forking
  fork
    setup_ints.start(m_sequencer); // Main bus activity
    begin
      forever begin
        fork // Waiting for one or more of 4 interrupts
          i_cfg.wait_for_IRQ0();
          i_cfg.wait_for_IRQ1();
          i_cfg.wait_for_IRQ2();
          i_cfg.wait_for_IRQ3();
        join_any
        disable fork;
        ISR.start(m_sequencer); // Start the ISR
      end
    end
    join_any // At the end of the main bus activity sequence
    disable fork;
  endtask: body

endclass: int_test_seq

```

Inside the ISR, the first action in the body method is the grab(), and the last action is ungrab(). If the ungrab() call was made earlier in the ISR sequence, then the main processing sequence would be able to resume sending sequence\_items to the bus interface.

```

// 
// Interrupt service routine
// 
// Looks at the interrupt sources to determine what to do
// 

```

```
class isr extends uvm_sequence #(bus_seq_item);  
  
`uvm_object_utils(isr)  
  
function new (string name = "isr");  
  super.new(name);  
endfunction  
  
rand logic[31:0] addr;  
rand logic[31:0] write_data;  
rand bit read_not_write;  
rand int delay;  
  
bit error;  
logic[31:0] read_data;  
  
task body;  
  bus_seq_item req;  
  
  m_sequencer.grab(this); // Grab => Immediate exclusive access to sequencer  
  
  req = bus_seq_item::type_id::create("req");  
  
  // Read from the GPO register to determine the cause of the interrupt  
  assert (req.randomize() with {addr == 32'h0100_0000; read_not_write == 1;});  
  start_item(req);  
  finish_item(req);  
  
  // Test the bits and reset if active  
  //  
  // Note that the order of the tests implements a priority structure  
  //  
  req.read_not_write = 0;  
  if(req.read_data[0] == 1) begin  
    `uvm_info("ISR:BODY", "IRQ0 detected", UVM_LOW)  
    req.write_data[0] = 0;  
    start_item(req);  
    finish_item(req);  
    `uvm_info("ISR:BODY", "IRQ0 cleared", UVM_LOW)  
  end  
  if(req.read_data[1] == 1) begin  
    `uvm_info("ISR:BODY", "IRQ1 detected", UVM_LOW)  
    req.write_data[1] = 0;  
    start_item(req);
```

```

    finish_item(req);
    `uvm_info("ISR:BODY", "IRQ1 cleared", UVM_LOW)
end
if(req.read_data[2] == 1) begin
    `uvm_info("ISR:BODY", "IRQ2 detected", UVM_LOW)
    req.write_data[2] = 0;
    start_item(req);
    finish_item(req);
    `uvm_info("ISR:BODY", "IRQ2 cleared", UVM_LOW)
end
if(req.read_data[3] == 1) begin
    `uvm_info("ISR:BODY", "IRQ3 detected", UVM_LOW)
    req.write_data[3] = 0;
    start_item(req);
    finish_item(req);
    `uvm_info("ISR:BODY", "IRQ3 cleared", UVM_LOW)
end
start_item(req); // Take the interrupt line low
finish_item(req);

m_sequencer.ungrab(this); // Ungrab the sequencer, let other sequences in

endtask: body

endclass: isr

```

Note that the way in which this ISR has been coded allows for a degree of prioritisation since each IRQ source is tested in order from IRQ0 to IRQ3.

( [download source code examples online at http://verificationacademy.com/uvm-ovm](http://verificationacademy.com/uvm-ovm) ).

## Prioritised ISR Sequence

A less disruptive approach to implementing interrupt handling using sequences is to use *sequence prioritisation*. Here, the interrupt monitoring thread starts the ISR sequence with a priority that is higher than that of the main process. This has the potential to allow other sequences with a higher priority than the ISR to gain access to the sequencer. Note that in order to use sequence prioritisation, the sequencer arbitration mode needs to be set to SEQ\_ARB\_STRICT\_FIFO, SEQ\_ARB\_STRICT\_RAND or STRICT\_ARB\_WEIGHTED.

Prioritising ISR sequences also enables modelling of prioritised ISRs, i.e. the ability to be able to interrupt an ISR with a higher priority ISR. However, since sequences are functor objects rather than simple sub-routines, multiple ISR sequences can be active on a sequencer, all that prioritisation affects is their ability to send a sequence\_item to the driver. Therefore, whatever processing is happening in an ISR will still continue even if a higher priority ISR "interrupts" it, which means that sequence\_items from the first lower priority ISR could potentially get through to the driver.

The following code example demonstrates four ISR sequences which are started with different priorities, allowing a higher priority ISR to execute in preference to a lower priority ISR.

```
class int_test_seq extends uvm_sequence #(bus_seq_item);

`uvm_object_utils(int_test_seq)

function new (string name = "int_test_seq");
  super.new(name);
endfunction

task body;
  set_ints setup_ints; // Main sequence running on the bus
  ISR0, ISR1, ISR2, ISR3; // Interrupt service routines

  int_config i_cfg;

  setup_ints = set_ints::type_id::create("setup_ints");
  // ISR0 is the highest priority
  ISR0 = isr::type_id::create("ISR0");
  ISR0.id = "ISR0";
  ISR0.i = 0;
  // ISR1 is medium priority
  ISR1 = isr::type_id::create("ISR1");
  ISR1.id = "ISR1";
  ISR1.i = 1;
  // ISR2 is medium priority
  ISR2 = isr::type_id::create("ISR2");
  ISR2.id = "ISR2";
  ISR2.i = 2;
  // ISR3 is lowest priority
  ISR3 = isr::type_id::create("ISR3");
  ISR3.id = "ISR3";
  ISR3.i = 3;

  i_cfg = int_config::get_config(m_sequencer);

  // Set up sequencer to use priority based on FIFO order
  m_sequencer.set_arbitration(SEQ_ARB_STRICT_FIFO);

  // A main thread, plus one for each interrupt ISR
  fork
    setup_ints.start(m_sequencer);
    forever begin // Highest priority
      i_cfg.wait_for_IRQ0();
      ISR0.isr_no++;
      ISR0.start(m_sequencer, this, HIGH);
    end
  join
endtask
```

```

    end

    forever begin // Medium priority
        i_cfg.wait_for_IRQ1();
        ISR1.isr_no++;
        ISR1.start(m_sequencer, this, MED);
    end

    forever begin // Medium priority
        i_cfg.wait_for_IRQ2();
        ISR2.isr_no++;
        ISR2.start(m_sequencer, this, MED);
    end

    forever begin // Lowest priority
        i_cfg.wait_for_IRQ3();
        ISR3.isr_no++;
        ISR3.start(m_sequencer, this, LOW);
    end

    join_any
    disable fork;

endtask: body

endclass: int_test_seq

```

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

## Interrupt Driven Sequence Execution

Some types of hardware systems use interrupts to schedule events or to control the order of processing. For instance, DSP based systems often consist of a chain of hardware acceleration blocks that take the contents of a memory block, perform a transform on it and write the results to another block of memory generating an interrupt when the transform is complete. Interrupts are used because the length of time to complete the transform is variable and it is not safe to start the next stage in the processing chain until the one before has completed. Each DSP hardware accelerator generates an interrupt when it is ready to process, or when it has completed a transform. This is a different situation from the previous examples, because the sequence execution is reliant on the interrupts coming in:

```

// DSP Control Sequence
//
// Each DSP processor in the chain generates an interrupt
// when it has completed processing
//
// The sequence then starts the next accelerator
//
class dsp_con_seq extends uvm_sequence #(dsp_con_seq_item);
    `uvm_object_utils(dsp_con_seq)

```

```
dsp_con_config cfg;
dsp_con_seq_item req;

function new(string name = "dsp_con_seq");
  super.new(name);
endfunction

// Wait for the interrupts to fire
// then start up the next DSP block
task body;

  cfg = dsp_con_config::get_config(m_sequencer);

  req = dsp_con_seq_item::type_id::create("req");

  cfg.wait_for_reset;
  repeat(2) begin
    do_go(4'h1); // Start Accelerator 0
    cfg.wait_for_irq0; // Accelerator 0 complete
    do_go(4'h2); // Start Accelerator 1
    cfg.wait_for_irq1; // Accelerator 1 complete
    do_go(4'h4); // Start Accelerator 2
    cfg.wait_for_irq2; // Accelerator 2 complete
    do_go(4'h8); // Start Accelerator 3
    cfg.wait_for_irq3; // Accelerator 3 complete
  end
  cfg.wait_for_clock;

endtask: body

// Toggles the go or start bit
task do_go(bit[3:0] go);
  req.go = go;
  start_item(req);
  finish_item(req);
endtask

endclass:dsp_con_seq
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>**  ).

# Sequences/Stopping

---

## Once started, sequences should not be stopped.

There are two methods available in the sequence and sequencer API that allow sequences to be killed. However, neither method checks that the driver is not currently processing any sequence\_items and the result is that any item\_done() or put() method called from the driver will either never reach the controlling sequence or will cause an UVM fatal error to occur because the sequences return pointer queue has been flushed.

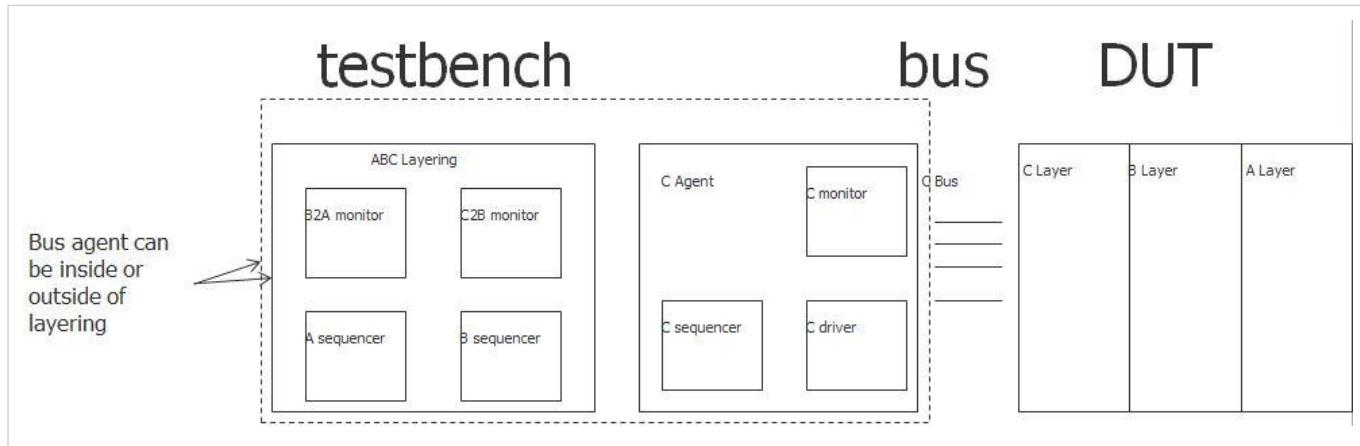
The methods are:

- <sequence>.kill()
- <sequencer>.stop\_sequences()

Do not use these methods unless you have some way of ensuring that the driver is inactive before making the call.

# Sequences/Layering

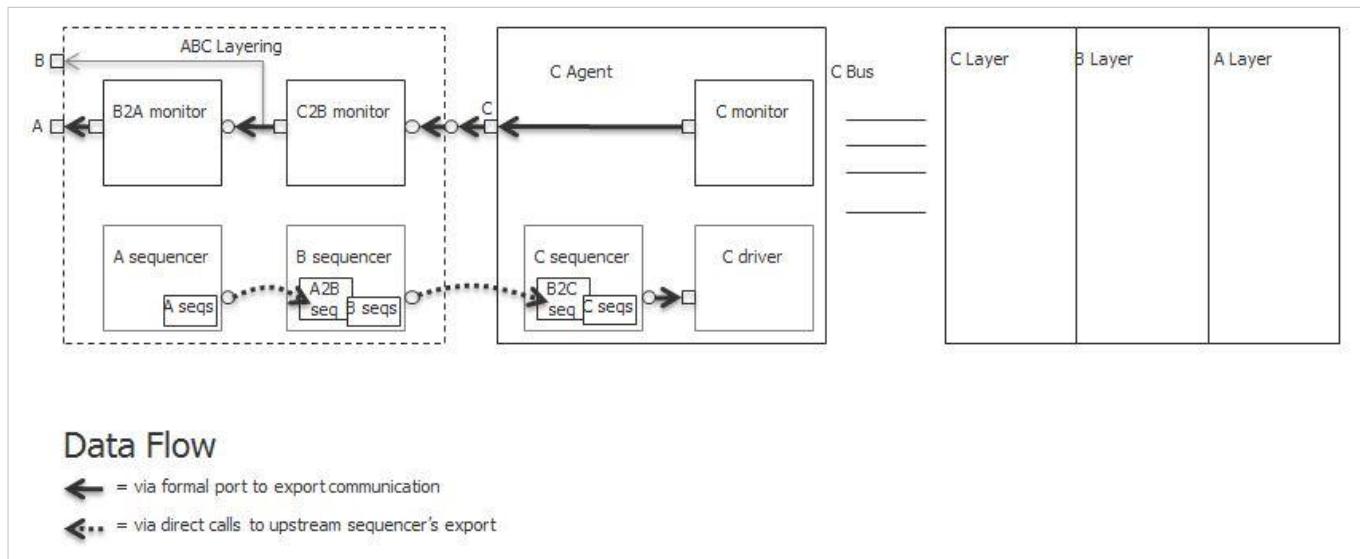
Many protocols have a hierarchical definition - for example, PCI express, USB 3.0, and MIPI LLI all have a Transaction Layer, a Transport Layer, and a Physical Layer. Sometimes we want to create a protocol independent layer on top of a standard protocol so that we can create protocol independent verification components ( for example TLM 2.0 GP over AMBA AHB ). All these cases require that we deconstruct sequence items of the higher level protocol into sequence items of the lower level protocol in order to stimulate the bus and that we reconstruct higher level sequence items from the lower level sequence items in the analysis datapath.



## The Architecture of a Layering

In order to do this we construct a layering component. A layering component:

- Must include a child sequencer for each non leaf level in the layering.
- Must create, connect and start a translator sequence for each non leaf level in the layering.
- Must have a handle to the leaf level protocol agent. This protocol agent may be a child of the layering or external to it.
- May include a reconstruction monitor for each non leaf level in the layering.
- Should create and connect external analysis ports for each monitor contained within the layering
- Will usually have a configuration object associated with it that contains the configuration objects of all the components contained within it.



## Child Sequencers

A child sequencer in a layering is simply the usual sequencer for that protocol. Very often an appropriately parameterized uvm\_sequencer is quite sufficient. If the higher level protocol has been modeled as a protocol UVC, then the layering should instantiate an instance of the sequencer used by the agent for that protocol so that sequences can be targeted either at the bus agent or the layering.

For example, the ABC layering below has an A\_sequencer and a B\_sequencer.

```
class ABC_layering extends uvm_subscriber #( C_item );
  `uvm_component_utils( ABC_layering )

  ...
  A_sequencer a_sequencer;
  B_sequencer b_sequencer;
  ...
  C_agent c_agent;

  function new(string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    a_sequencer = A_sequencer::type_id::create("a_sequencer",this);
    b_sequencer = B_sequencer::type_id::create("b_sequencer",this);
  endfunction
  ...
endclass
```

## Translator Sequences

A sequence which translates from upstream items to downstream items runs on the downstream sequencer but has a reference to the upstream sequencer. It directly references the upstream sequencer to call `get_next_item` and `item_done` to get upstream items and tell the upstream sequencer that it has finished processing the upstream sequence item. Between `get_next_item` and `item_done` it sends data to and gets data from the lower level sequencer by calling `start_item` and `finish_item`. A simple BtoC translator sequence is shown below:

```
class BtoC_seq extends uvm_sequence #(C_item);
  `uvm_object_utils(BtoC_seq);

  function new(string name="";
  super.new(name);
  endfunction

  uvm_sequencer #(B_item) up_sequencer;

  virtual task body();
    B_item b;
    C_item c;
    int i;
    forever begin
      up_sequencer.get_next_item(b);
      foreach (b.fb[i]) begin
        c = C_item::type_id::create();
        start_item(c);
        c.fc = b.fb[i];
        finish_item(c);
      end
      up_sequencer.item_done();
    end
  endtask
endclass
```

The run phase of the ABC\_layering component is responsible for creating the translator sequences, connecting them to their upstream sequencers, and starting them on their downstream sequencers:

```
virtual task run_phase(uvm_phase phase);
  AtoB_seq a2b_seq;
  BtoC_seq b2c_seq;

  a2b_seq = AtoB_seq::type_id::create("a2b_seq");
  b2c_seq = BtoC_seq::type_id::create("b2c_seq");

  // connect translation sequences to their respective upstream sequencers
```

```

a2b_seq.up_sequencer = a_sequencer;
b2c_seq.up_sequencer = b_sequencer;

// start the translation sequences
fork
  a2b_seq.start(b_sequencer);
  b2c_seq.start(c_agent.c_sequencer);
join_none
endtask

```

## The Protocol Agent

Every layering must have a handle to the leaf level protocol agent. If we are delivering verification IP for a layered protocol, it usually makes sense to deliver the layering with an internal protocol agent. On the other hand, we may be adding a layering for use with a shrink wrapped protocol agent instantiated elsewhere in the testbench. Under these circumstances we will want the leaf level protocol agent to be outside the layering.

### Internal Protocol Agent

In the case of an internal protocol agent, the layering component inherits from `uvm_component` and creates a child layering agent:

```

class ABC_layering extends uvm_component;
  `uvm_component_utils( ABC_layering )

  ...
  A_sequencer a_sequencer;
  B_sequencer b_sequencer;
  ...
  C_agent c_agent;

  function new(string name, uvm_component parent=null);
    super.new(name, parent);
  endfunction

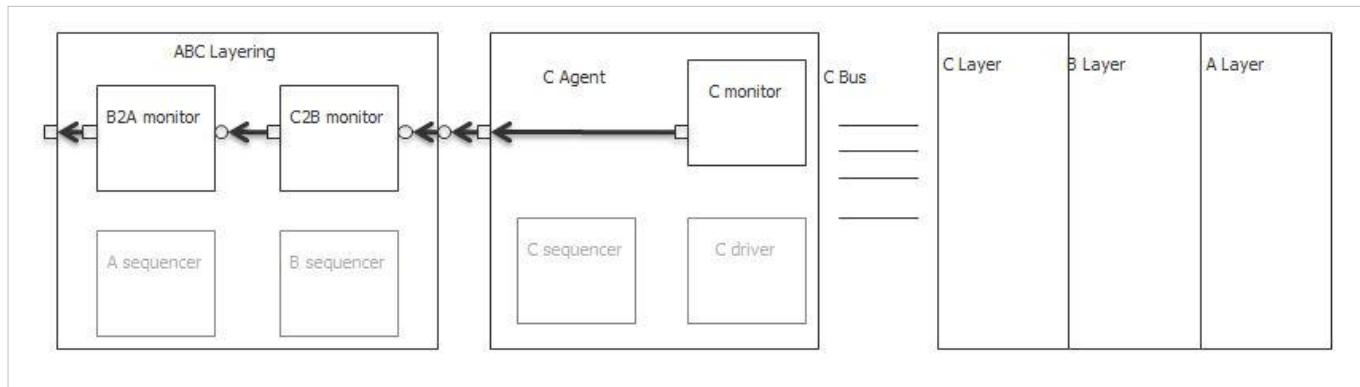
  function void build_phase(uvm_phase phase);
    a_sequencer = A_sequencer::type_id::create("a_sequencer",this);
    b_sequencer = B_sequencer::type_id::create("b_sequencer",this);
    c_agent = C_agent::type_id::create("c_sequencer",this);
  endfunction
  ...
endclass

```

## External Protocol Agent

In the case of an external protocol agent, the layering is a subscriber parameterized on the leaf level sequence item and the agent is not constructed inside the layering. The code introduced above shows the code for an external agent.

## The Analysis Path



Really, there is nothing special in the analysis path of a layering. For each layer in the monitoring we provide a reconstruction monitor which assembles high level items from low level items. These reconstruction monitors have an analysis export which is connected to the analysis ports of the lower level monitor and an analysis port. This analysis port is connected to an external analysis port and the analysis export of the upstream monitor if there is one.

An outline of a reconstruction monitor is shown below:

```
class C2B_monitor extends uvm_subscriber #(C_item); // provides an analysis export of type C_item
`uvm_component_utils(C2B_monitor)

uvm_analysis_port#(B_item) ap;
// declarations omitted ...

function new(string name, uvm_component parent);
  super.new(name, parent);
  ap = new("ap",this);
endfunction

function void write(C_item t);
  // reconstruction code omitted ...
  ap.write( b_out );
  ...
endfunction
endclass
```

The reconstruction monitors are connected up in the normal way:

```
class ABC_layering extends uvm_subscriber #( C_item );
`uvm_component_utils( ABC_layering )
```

```
uvm_analysis_port #( A_item ) ap;

A_sequencer a_sequencer;
B_sequencer b_sequencer;

C2B_monitor c2b_mon;
B2A_monitor b2a_mon;

C_agent c_agent;

function new(string name, uvm_component parent=null);
    super.new(name, parent);
endfunction

function void build_phase(uvm_phase phase);
    a_sequencer = A_sequencer::type_id::create("a_sequencer",this);
    b_sequencer = B_sequencer::type_id::create("b_sequencer",this);

    c2b_mon = C2B_monitor::type_id::create("c2b_mon",this);
    b2a_mon = B2A_monitor::type_id::create("b2a_mon",this);

    ap = new("ap" , this );
endfunction

function void connect_phase(uvm_phase phase);
    c2b_mon.ap.connect(b2a_mon.analysis_export);
    b2a_mon.ap.connect( ap );
endfunction

...
endclass
```

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

---

# Register Abstraction Layer

---

## Registers

---

Learn all about methodology related to UVM Register Package

---

### Registers Chapter contents:

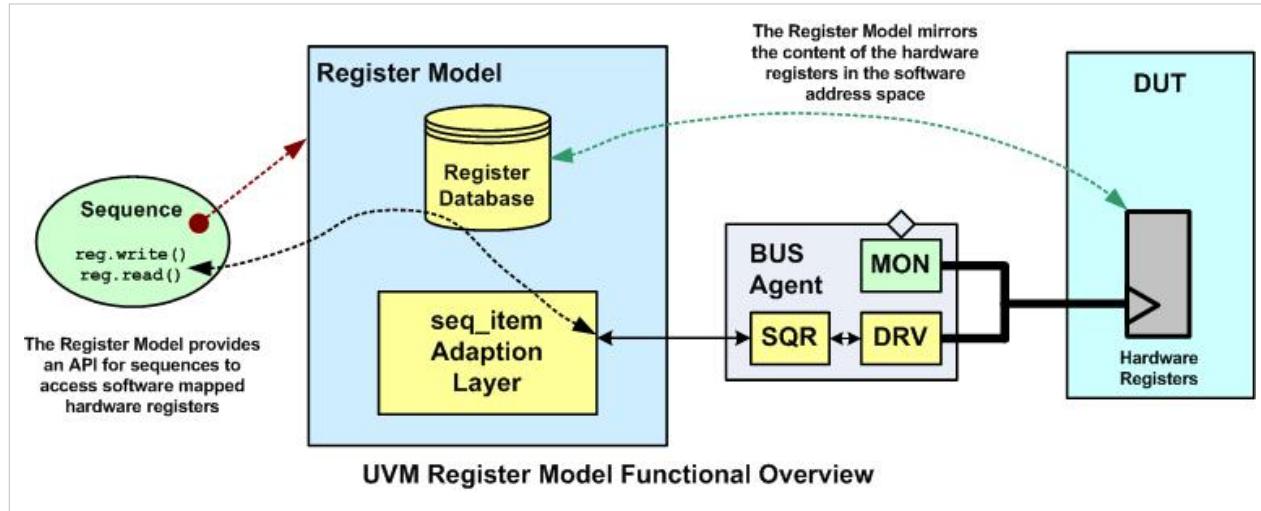
Registers (this page)  
Registers/Specification  
Registers/Adapter  
Registers/AdapterContext  
Registers/Integrating  
Registers/Integration  
Registers/RegisterModelOverview  
Registers/ModelStructure  
Registers/QuirkyRegisters  
Registers/ModelCoverage  
Registers/BackdoorAccess  
Registers/Generation  
Registers/StimulusAbstraction  
Registers/MemoryStimulus  
Registers/SequenceExamples  
Registers/BuiltInSequences  
Registers/Configuration  
Registers/Scoreboarding  
Registers/FunctionalCoverage

---

## Topic Overview

### Introduction

The UVM register model provides a way of tracking the register content of a DUT and a convenience layer for accessing register and memory locations within the DUT.

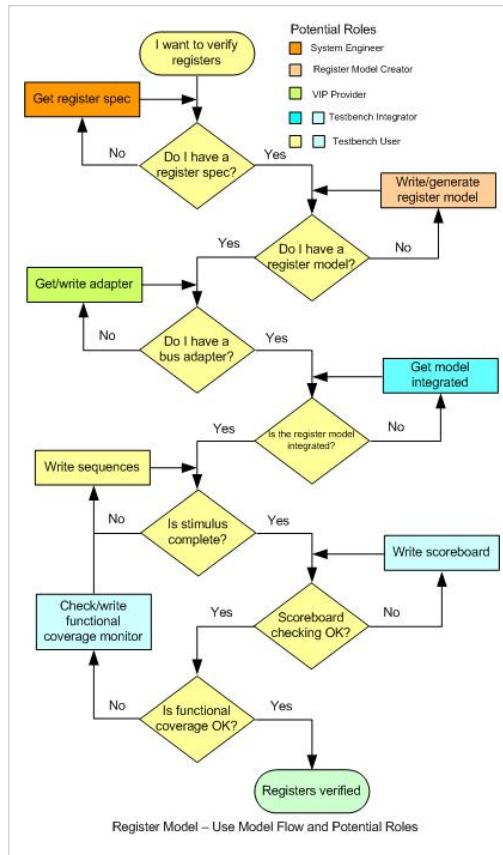


The register model abstraction reflects the structure of a hardware-software register specification, since that is the common reference specification for hardware design and verification engineers, and it is also used by software engineers developing firmware layer software. It is very important that all three groups reference a common specification and it is crucial that the design is verified against an accurate model.

The UVM register model is designed to facilitate productive verification of programmable hardware. When used effectively, it raises the level of stimulus abstraction and makes the resultant stimulus code straight-forward to reuse, either when there is a change in the DUT register address map, or when the DUT block is reused as a sub-component.

## How The UVM Register Material Is Organised

The UVM register model can be considered from several different viewpoints and this page is separated into different sections so that you can quickly navigate to the material that concerns you most. The following diagram summarises the various steps in the flow for using the register model and outlines the different categories of users.



Therefore, the different register viewpoints are:

- The VIP developer
- The Register Model writer
- The Testbench Integrator
- The Testbench User

### VIP Developer Viewpoint

In order to support the use of the UVM register package, the developer of an On Chip Bus verification component needs to develop an adapter class. This adapter class is responsible for translating between the UVM register packages generic register sequence\_items and the VIP specific sequence\_items. Developing the adapter requires knowledge of the target bus protocol and how the different fields in the VIP sequence\_item relate to that protocol.

Once the adapter is in place it can be used by the testbench developer to integrate the register model into the UVM testbench.

To understand how to create an adapter the suggested route through the register material is:

Step	Page	Description	Relevance
1	Integrating	Describes how the adaptor fits into the overall testbench architecture	Background
2	integration	Describes in detail how the adaptor is used	Background
3	Adapter	How to implement a register adaptor, with an example	Essential
4	AdapterContext	How to provide context to an adapter, with an example	Generally Unnecessary

## Creating A Register Model

A register model can be created using a register generator application or it can be written by hand. In both cases, the starting point is the hardware-software register specification and this is transformed into the model.

If you are using a generator or writing a register model based on a register specification then these topics should be followed in this order:

			Relevance	
Step	Page	Description	Using Generator	Writing Model
1	Specification	Overview of Register Specification	Background	Background
2	RegisterModelOverview	Register Model Hierarchy Overview	Useful background	Essential
3	ModelStructure	Implementing a register model	Background	Essential
4	QuirkyRegisters	Implementing 'Quirky' registers	Essential	Essential
5	ModelCoverage	Adding coverage models	Background	Essential
6	BackdoorAccess	Using and specifying back door accesses	Background	Essential
7	Generation	Generating a register model	Essential	Unnecessary

## Integrating A Register Model

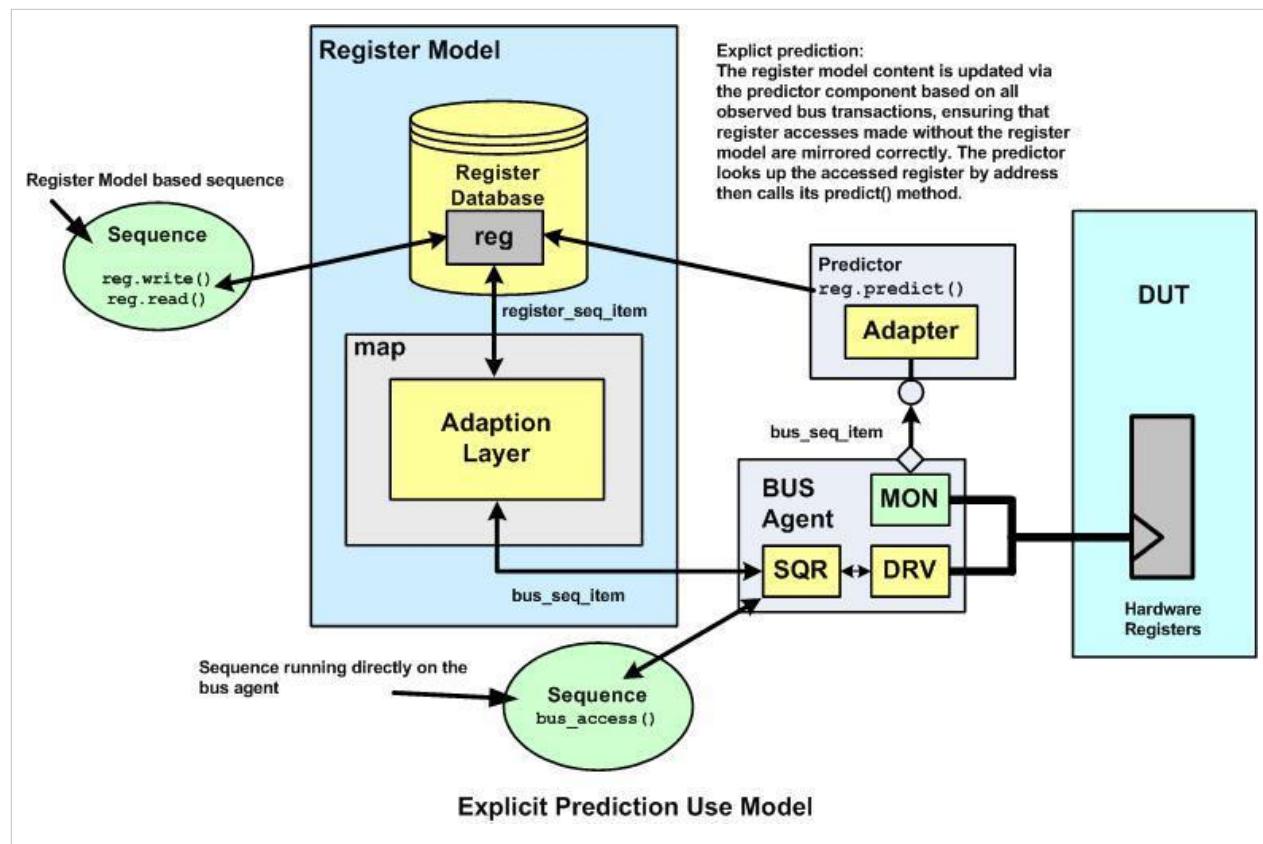
### Integration Pre-requisites

If you are integrating a register model into a testbench, then the pre-requisites are that a register model has been written and that there is an adaptor class available for the bus agent that is going to be used to interact with the DUT bus interface.

### Integration Process

In the testbench, the register model object needs to be constructed and a handle needs to be passed around the testbench environment using either the [configuration](#) and/or the [resource](#) mechanism.

In order to drive an agent from the register model an association needs to be made between it and the target sequencer so that when a sequence calls one of the register model methods a bus level sequence\_item is sent to the target bus driver. The register model is kept updated with the current hardware register state via the bus agent monitor, and a predictor component is used to convert bus agent analysis transactions into updates of the register model.



The testbench integrator might also be involved with implementing other analysis components which reference the register model, and these would include a scoreboard and a functional coverage monitor.

For the testbench integrator, the recommended route through the register material is outlined in the table below:

Step	Page	Description	Relevance
1	RegisterModelOverview	Overview of the register model hierarchy	Useful to understand terminology
2	Integrating	Overview of the register model stimulus and prediction architecture	Essential
3	Adapter	Adapter Implementation detail	Useful background
4	AdapterContext	How to provide context to an adapter, with an example	Useful background
5	integration	Register model integration detail	Essential
6	Scoreboarding	Scoreboard implementation	Useful background
7	FunctionalCoverage	Coverage implementation	Useful background

## Using A Register Model

Once it has been integrated, the register model is used by the testbench user to create stimulus using sequences or through analysis components such as scoreboards and functional coverage monitors.

The register model is intended to make it easier to write reusable sequences that access hardware registers and areas of memory. The model data structure is organised to reflect the DUT hierarchy and this makes it easier to write abstract and reusable stimulus in terms of hardware blocks, memories, registers and fields rather than working at a lower bit pattern level of abstraction. The model contains a number of access methods which sequences use to read and write registers. These methods cause generic register transactions to be converted into transactions on the target bus.

The UVM package contains a library of built-in test sequences which can be used to do most of the basic register and memory tests, such as checking register reset values and checking the register and memory data paths. These tests can be disabled for those areas of the register or memory map where they are not relevant using register attributes.

One common form of stimulus is referred to as configuration. This is when a programmable DUT has its registers set up to support a particular mode of operation. The register model can support auto-configuration, a process whereby the contents of the register model are forced into a state that represents a device configuration using constrained randomization and then transferred into the DUT.

The register model supports front door and back door access to the DUT registers. Front door access uses the bus agent in the testbench and register accesses use the normal bus transfer protocol. Back door access uses simulator data base access routines to directly force or observe the register hardware bits in zero time, by-passing the normal bus interface logic.

As a verification environment evolves, users may well develop analysis components such as scoreboards and functional coverage monitors which refer to the contents of the register model in order to check DUT behaviour or to ensure that it has been tested in all required configurations.

If you are a testbench consumer using the register model, then you should read the following topics in the recommended order:

Step	Page	Description	Relevance
1	Specification	Register Specification	Background
2	RegisterModelOverview	Register Model Hierarchy Overview	Essential to understand the terminology
3	Integrating	Register Model Testbench architecture	Background
4	StimulusAbstraction	Stimulus Abstraction for registers	Essential
5	MemoryStimulus	Memory stimulus abstraction	Essential
6	BackdoorAccess	Back door accesses	Relevant if you need to do backdoor accesses
7	SequenceExamples	Example sequences	Essential
8	Configuration	How to configure a programmable DUT	Essential
9	BuiltInSequences	How to use the UVM built-in register sequences	May be relevant
10	Scoreboarding	Implementing a register model based scoreboard	Important if you need to maintain a scoreboard.
11	FunctionalCoverage	Implementing functional coverage using the register model	Important if you need to enhance a functional coverage model

## Register Model Examples

The UVM register use model is illustrated by code excerpts which are taken from two example testbenches. The main example is a complete verification environment for a SPI master DUT, in addition to register model this includes a scoreboard and a functional coverage monitor, along with a number of test cases based on the use of register based sequences. The other example is designed to illustrate the use of memories and some of the built-in register sequences from the UVM library. Download links for these examples are provided in the table below:

Example	Download Link
SPI Master Testbench	( <a href="#">download source code examples online at [1]</a> ).
Memory Sub-System Testbench	( <a href="#">download source code examples online at [1]</a> ).

# Registers/Specification

Hardware functional blocks connected to host processors are managed via memory mapped registers. This means that each bit in the software address map corresponds to a hardware flip-flop. In order to control and interact with the hardware, software has to read and write the registers and so the register description is organised using an abstraction which is referred to as the hardware-software interface, or as the register description.

This hardware-software interface allocates addresses in the I/O memory map to a register which is identified by a mnemonic. Each register may then be broken down into fields, or groups of individual bits which again are given a mnemonic. A field may just be a single bit, or it may be as wide as the register itself. Each field may have different access attributes, for instance it might be read-only, write-only, read-write or clear on read. The register may have reserved fields, in other words bits in the register which are not used but might be used in future versions of the design. Here is an example of a register description - the control register from an SPI master DUT.

## SPI Control Register - Reset Value = 0

Bit Pos	31:14	13	12	11	10	9	8	7	6:0
Access	RO	R/W	R/W	R/W	R/W	R/W	R/W	RO	R/W
Name	Reserved	ACS	IE	LSB	Tx_NEG	Rx_NEG	GO_BSY	Reserved	CHAR_LEN

For a given functional block, there may be multiple registers and each one will have an address which is offset from the base address of the block. To access a register, a processor will do a read or write to its address. The software engineer uses the register map to determine how to program a hardware device, and he may well use a set of define statements which map register names and field names to numeric values in a header file so that he can work at a more abstract level of detail. For instance, if he wishes to enable interrupts in the SPI master he may do a read from CTRL, then OR the value with IE and then write back the resultant value to CTRL:

```
spi_ctrl = reg_read(CTRL);
spi_ctrl = spi_ctrl | IE;
reg_write(CTRL, spi_ctrl);
```

The register address map for the SPI master is as in the table below. Note that the control register is at an offset address of 0x10 from the SPI master base address.

## SPI Master Register Address Map

Name	Address Offset	Width	Access	Description
RX0	0x00	32	RO	Receive data register 0
TX0	0x00	32	WO	Transmit data register 0
RX1	0x04	32	RO	Receive data register 1
TX1	0x04	32	WO	Transmit data register 1
RX2	0x08	32	RO	Receive data register 2
TX2	0x08	32	WO	Transmit data register 2
RX3	0x0c	32	RO	Receive data register 3
TX3	0x0c	32	WO	Transmit data register 3
CTRL	0x10	32	R/W	Control and status register
DIVIDER	0x14	32	R/W	Clock divider register
SS	0x18	32	R/W	Slave Select Register

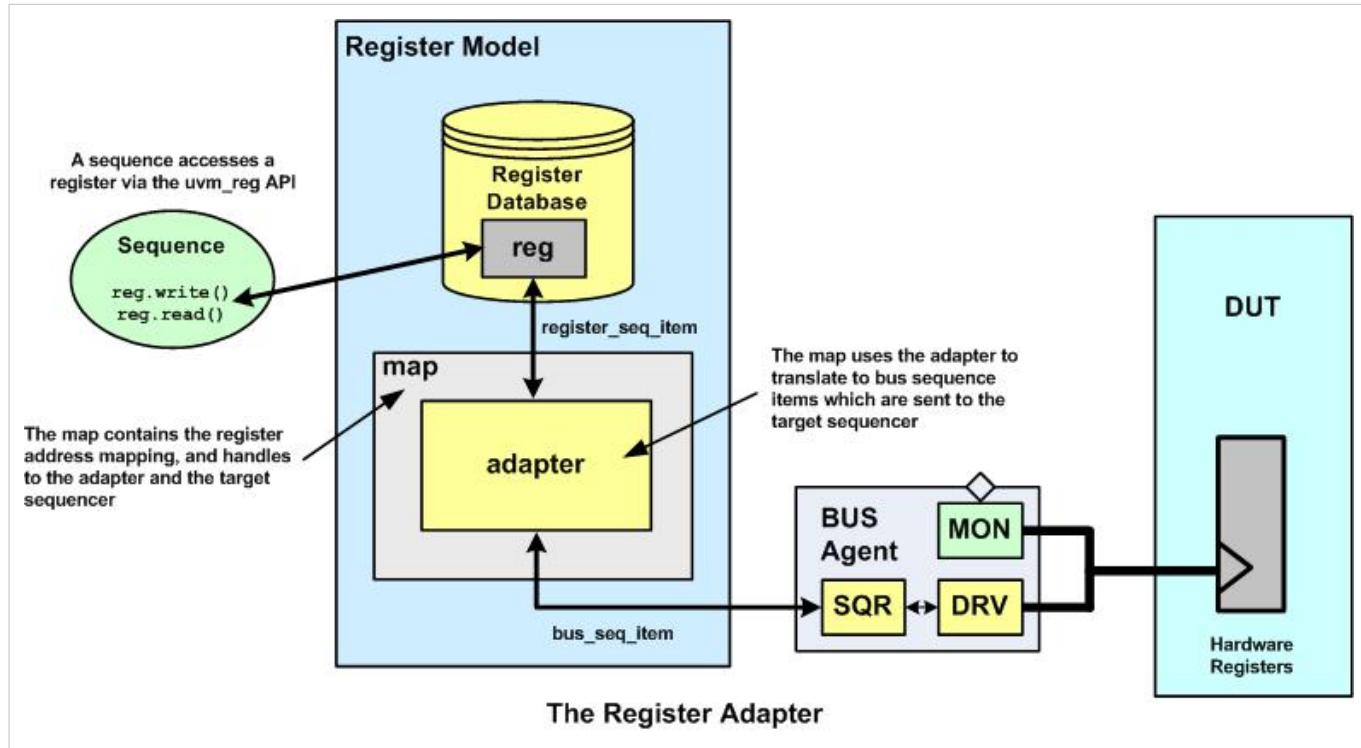
**Note:** In the SPI master the RXn and TXn registers actually map onto the same hardware flip-flops and during character transmission and reception their content changes. Therefore from the software interface abstraction the TX registers are write only, and the RX registers are read only, and the Rx content is only valid at the end of a character transfer.

If the SPI master is integrated into a larger design then the base address of the SPI master will change, the software engineer protects himself from the effects of that change by updating his header file, but the underlying firmware code does not need to change.

# Registers/Adapter

## Overview

The UVM register model access methods generate bus read and write cycles using generic register transactions. These transactions need to be adapted to the target bus sequence\_item. The adapter needs to be bidirectional in order to convert register transaction requests to bus sequence items, and to be able to convert bus sequence item responses back to bus sequence items. The adapter should be implemented by extending the uvm\_reg\_adapter base class.



## Adapter Is Part Of An Agent Package

The register adapter should be supplied as part of the target bus agent package, but since the UVM register model capability is relatively new you will most likely have to implement your own register adapter until the agent provider incorporates one into the agent package.

If you are creating your own bus agent, then you should include an adapter to allow users of your agent package to be able to use it with the register model.

## Implementing An Adapter

The generic register item is implemented as a struct in order to minimise the amount of memory resource it uses. The struct is defined as type `uvm_reg_bus_op` and this contains 6 fields:

Property	Type	Comment/Description
addr	<code>uvm_reg_addr_t</code>	Address field, defaults to 64 bits
data	<code>uvm_reg_data_t</code>	Read or write data, defaults to 64 bits
kind	<code>uvm_access_e</code>	<code>UVM_READ</code> or <code>UVM_WRITE</code>
n_bits	<code>unsigned int</code>	Number of bits being transferred
byte_en	<code>uvm_reg_byte_en_t</code>	Byte enable
status	<code>uvm_status_e</code>	<code>UVM_IS_OK</code> , <code>UVM_IS_X</code> , <code>UVM_NOT_OK</code>

These fields need to be mapped to/from the target bus sequence item and this is done by extending the `uvm_reg_adapter` class which contains two methods - `reg2bus()` and `bus2reg()` which need to be overlaid. The adapter class also contains two property bits - `supports_byte_enable` and `provides_responses`, these should be set according to the functionality supported by the target bus and the target bus agent.

<b>uvm_reg_adapter</b>	
Methods	Description
<code>reg2bus</code>	Overload to convert generic register access items to target bus agent sequence items
<code>bus2reg</code>	Overload to convert target bus sequence items to register model items
Properties (Of type bit)	Description
<code>supports_byte_enable</code>	Set to 1 if the target bus and the target bus agent supports byte enables, else set to 0
<code>provides_responses</code>	Set to 1 if the target agent driver sends separate response sequence_items that require response handling

Taking the APB bus as a simple example; the bus sequence\_item, `apb_seq_item`, contains 3 fields (`addr`, `data`, `we`) which correspond to address, data, and bus direction. Address and data map directly and the APB item write enable maps to the bus item `kind` field. When converting an APB bus response to a register item the `status` field will be set to `UVM_IS_OK` since the APB agent does not support the `SLVERR` status bit used by the APB.

Since the APB bus does not support byte enables, the `supports_byte_enable` bit is set to 0 in the constructor of the APB adapter.

The `provides_responses` bit should be set if the agent driver returns a separate response item (i.e. `put(response)`, or `item_done(response)`) from its request item - see Driver/Sequence API. This bit is used by the register model layering code to determine whether to wait for a response or not, if it is set and the driver does not return a response, then the stimulus generation will lock-up.

Since the APB driver being used returns an item `done()`, it therefore uses a single item for both request and response, so the `provides_responses` bit is also set to 0 in the constructor.

The example code for the register to APB adapter is as follows:

```
class reg2apb_adapter extends uvm_reg_adapter;
  `uvm_object_utils(reg2apb_adapter)
```

```

function new(string name = "reg2apb_adapter");
    super.new(name);
    supports_byte_enable = 0;
    provides_responses = 0;
endfunction

virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
    apb_seq_item apb = apb_seq_item::type_id::create("apb");
    apb.we = (rw.kind == UVM_READ) ? 0 : 1;
    apb.addr = rw.addr;
    apb.data = rw.data;
    return apb;
endfunction: reg2bus

virtual function void bus2reg(uvm_sequence_item bus_item,
                             ref uvm_reg_bus_op rw);
    apb_seq_item apb;
    if (!$cast(apb, bus_item)) begin
        `uvm_fatal("NOT_APB_TYPE", "Provided bus_item is not of the correct type")
        return;
    end
    rw.kind = apb.we ? UVM_WRITE : UVM_READ;
    rw.addr = apb.addr;
    rw.data = apb.data;
    rw.status = UVM_IS_OK;
endfunction: bus2reg

endclass: reg2apb_adapter

```

The adapter code can be found in the file **reg2apb\_adapter.svh** in the */agents/apb\_agent* directory in the example which can be downloaded:

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

## Burst Support

The register model access methods support single accesses to registers and this is in line with the usual use model for register accesses - they are accessed individually, not as part of a burst. If you have registers which need to be tested for burst mode accesses, then the recommended approach is to initiate burst transfers from a sequence running directly on the target bus agent.

If you use burst mode to access registers, then the predictor implementation needs to be able to identify bursts and convert each beat into a predict() call on the appropriate register.

## Common Adapter Issues

The adapter is a critical link in the sequence communication chain between the register model access methods and the driver. The expectation is that the driver-sequence API is cleanly implemented and follows either a `get_next_item()/item_done()` or a `get()/put(rsp)` completion model on the driver side. The `provides_responses` bit should be set to 1 if your driver is using `put(rsp)` to return responses, and 0 if it does not.

If you get the setting of `provides_responses` wrong, then one of two things will happen - either the stimulus generation will lock up or you will get an immediate return from front door access methods with the wrong responses but see bus activity occur some time later.

The lock up occurs if you are using the `get_next_item()/item_done()` completion model and you have set the `provides_responses` bit to a 1 - the reason being that the adapter is waiting for a response that will never be returned.

The instant return occurs if you are using the `get()/put(rsp)` completion model and you have set the `provides_responses` bit to 0 and the `get()` call in the driver code immediately unblocks the sequencer completing the access method. The driver then goes on to execute the bus access before returning the response via the `put(rsp)` method. The adapter ignores the response.

You should also ensure that both read and write accesses follow the same completion model. In particular, make sure that you return a response from the driver for write accesses when using the `get()/put(rsp)` completion model.

## Context for the Bus

If the bus protocol targeted by the adapter needs more information to function properly, a method exists for providing context information to the adapter. Please visit the Adapter Context article for more information.

# Registers/Integrating

---

## Register Model Testbench Integration - Testbench Architecture Overview

Within an UVM testbench a register model is used either as a means of looking up a mirror of the current DUT hardware state or as means of accessing the hardware via the front or back door and updating the register model database.

For those components or sequences that use the register model, the register model has to be constructed and its handle passed around using a configuration object or a resource. Components and sequences are then able to use the register model handle to call methods to access data stored within it, or to access the DUT.

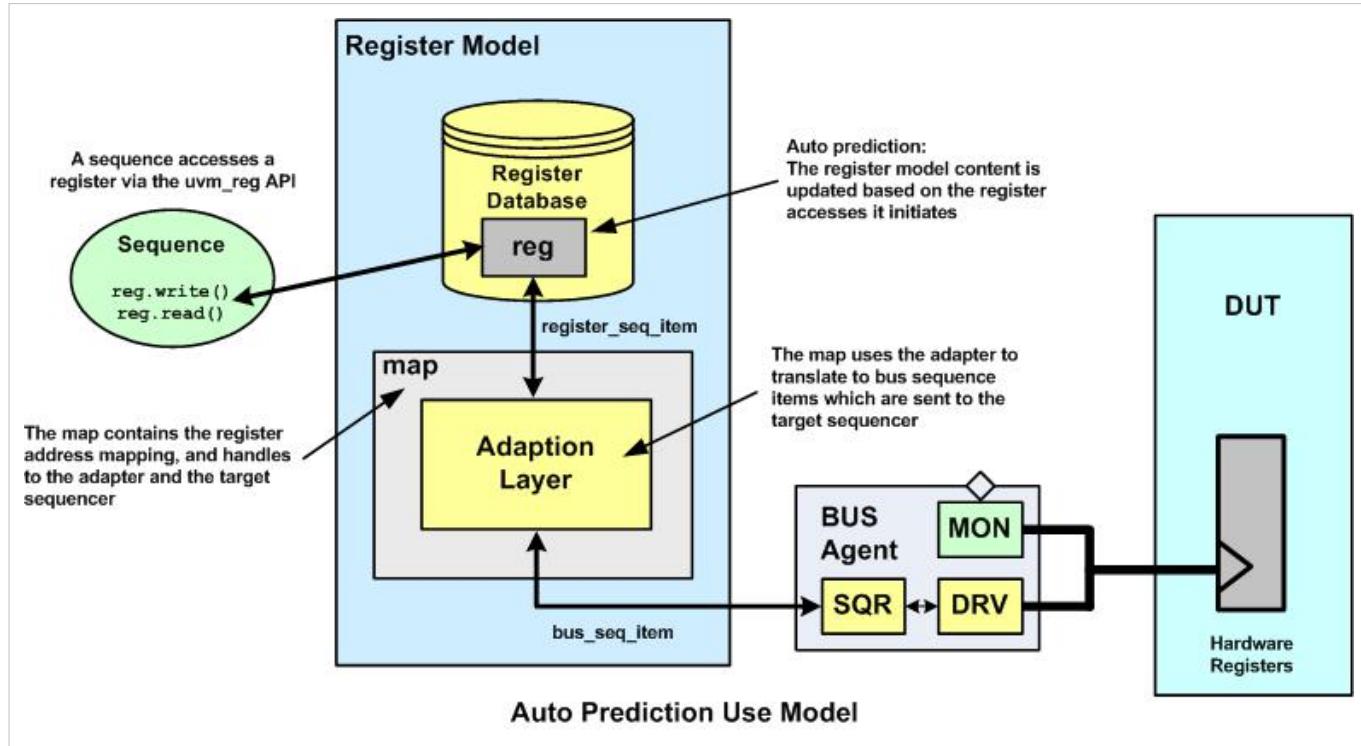
In order to make back door accesses to the DUT, the register model uses hdl paths which are used by simulator runtime database access routines to peek and poke the hardware signals corresponding to the register. The register model is updated automatically at the end of each back door access cycle. The way in which this update is done is by calling the predict() method which updates the accessed registers mirrored value. In order for back door accesses to work, no further integration with the rest of the testbench structure is required.

The register model supports front door accesses to the DUT by generating generic register transactions which are converted to target bus agent specific sequence\_items before being sent to the target bus agents sequencer, and by converting any returned response information back from bus agent sequence\_items into register transactions. This bidirectional conversion process takes place inside an adapter class which is specific to the target bus agent. There is really only way in which the stimulus side of the access is integrated with the testbench, but the update, or prediction, of the register model content at the end of a front door access can occur using one of three models and these are:

- Auto Prediction
- Explicit Prediction
- Passive Prediction

## Auto Prediction

Auto prediction is the simplest prediction regime for register accesses. In this mode, the various access methods which cause front door accesses to take place automatically call a predict() method using either the data that was written to the register, or the data read back from the register at the end of the bus cycle.

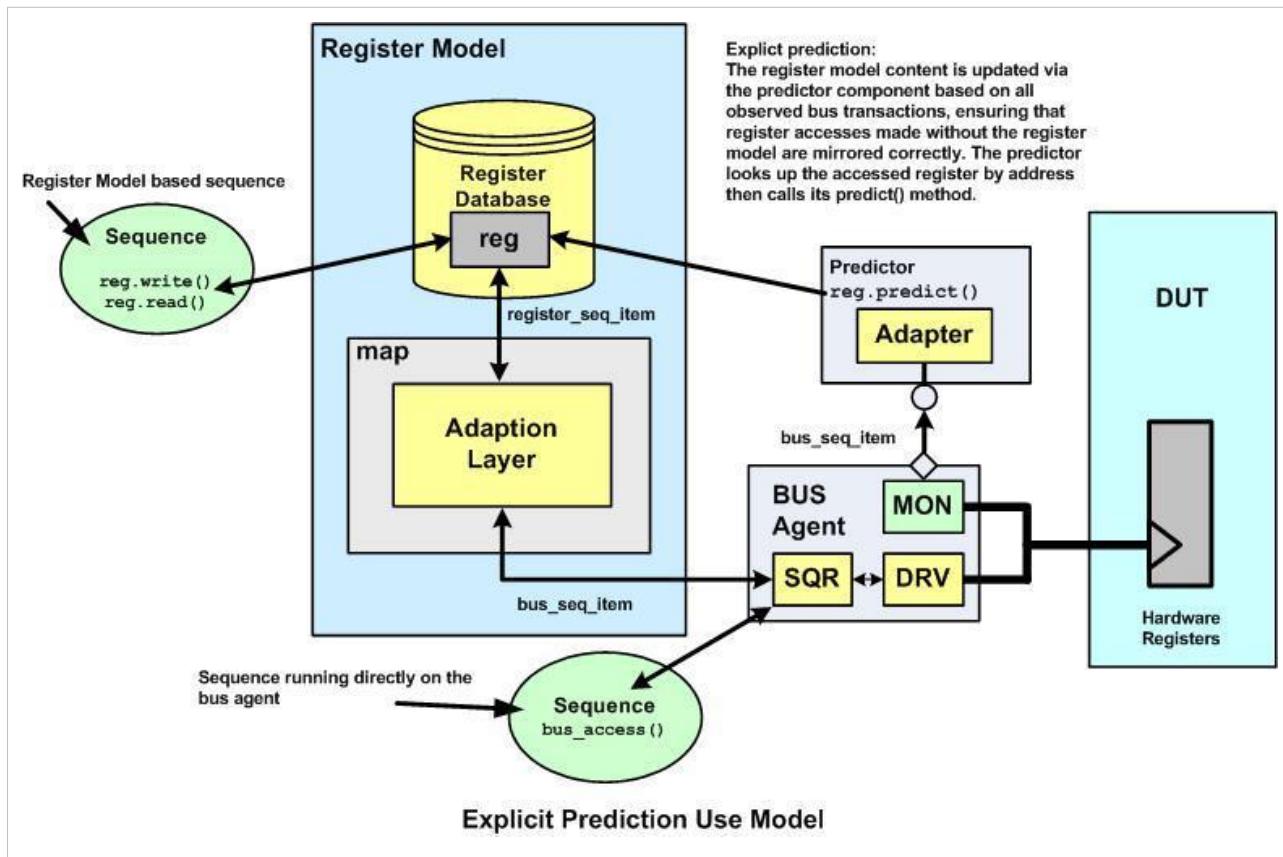


This mode of operation is the simplest to implement, but suffers from the drawback that it can only keep the register model up to date with the transfers that it initiates. If any other sequences directly access the target sequencer to update register content, or if there are register accesses from other DUT interfaces, then the register model will not be updated.

Explicit prediction is the default mode of prediction, to enable auto prediction, use the set\_auto\_predict() method. Also note that when using auto prediction, if the status returned is UVM\_NOT\_OK, the register model will not be updated.

## Explicit Prediction (Recommended Approach)

In the explicit prediction mode of operation an external predictor component is used to listen for target bus agent analysis transactions and then to call the predict() method of the accessed register to update its mirrored value. The predictor component uses the adapter to convert the bus analysis transaction into a register transaction, then uses the address field to look up the target register before calling its predict() method with the data field. Explicit prediction is the default mode of prediction, to disable it use the set\_auto\_predict() method.



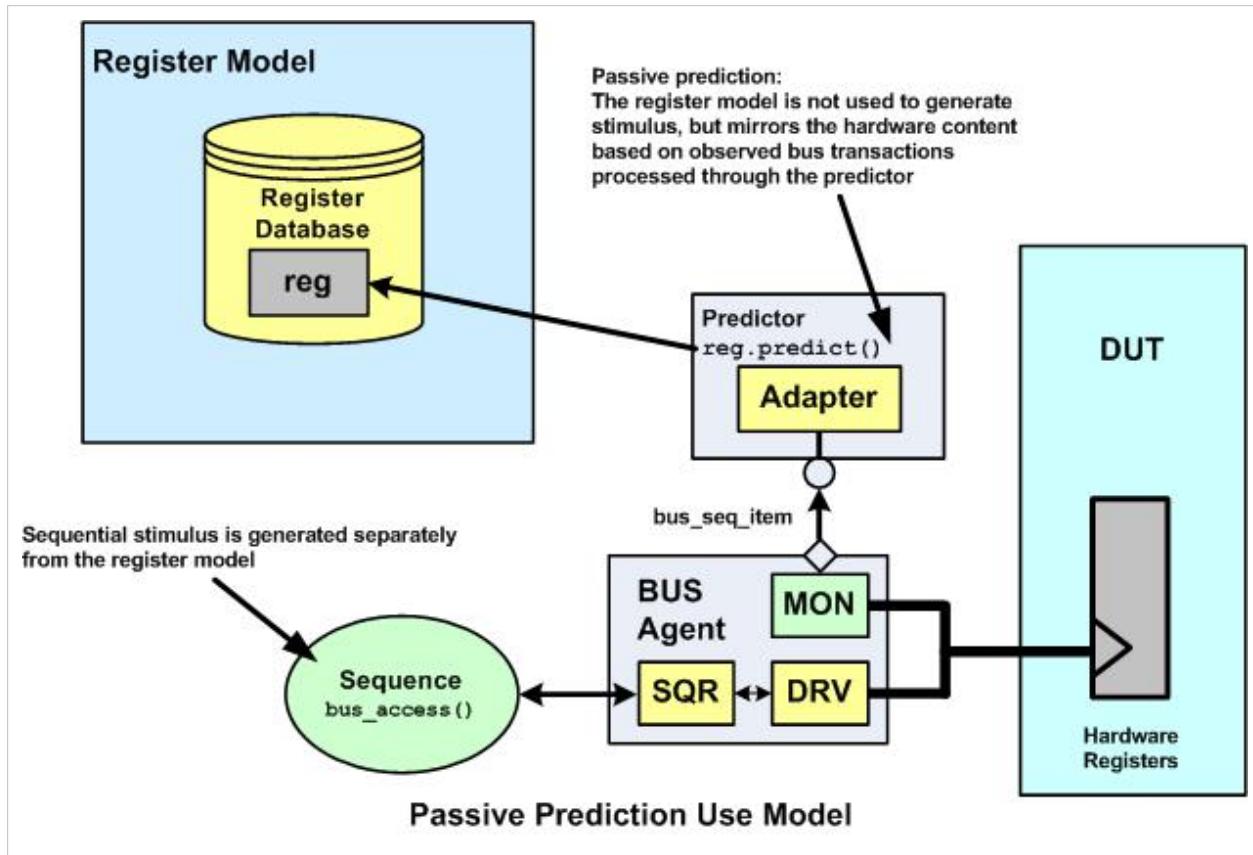
The main advantage of using explicit prediction is that it keeps the register model up to date with all accesses that occur on the target bus interface. The configuration also has more scope for supporting vertical reuse where accesses to the DUT may occur from other bus agents via bus bridges or interconnect fabrics.

During vertical reuse, an environment that supports explicit prediction can also support passive prediction as a result of re-configuration.

Also note that when using explicit prediction, the status value returned to the predictor is ignored. This means that if an errored (UVM\_NOT\_OK) status is being returned from a register access, the register access will need to be filtered out before sending information to the Predictor if that errored transfer is not meant to update the mirrored value of a register. This could be done in a monitor or with a modified testbench predictor component placed between a monitor and a Predictor.

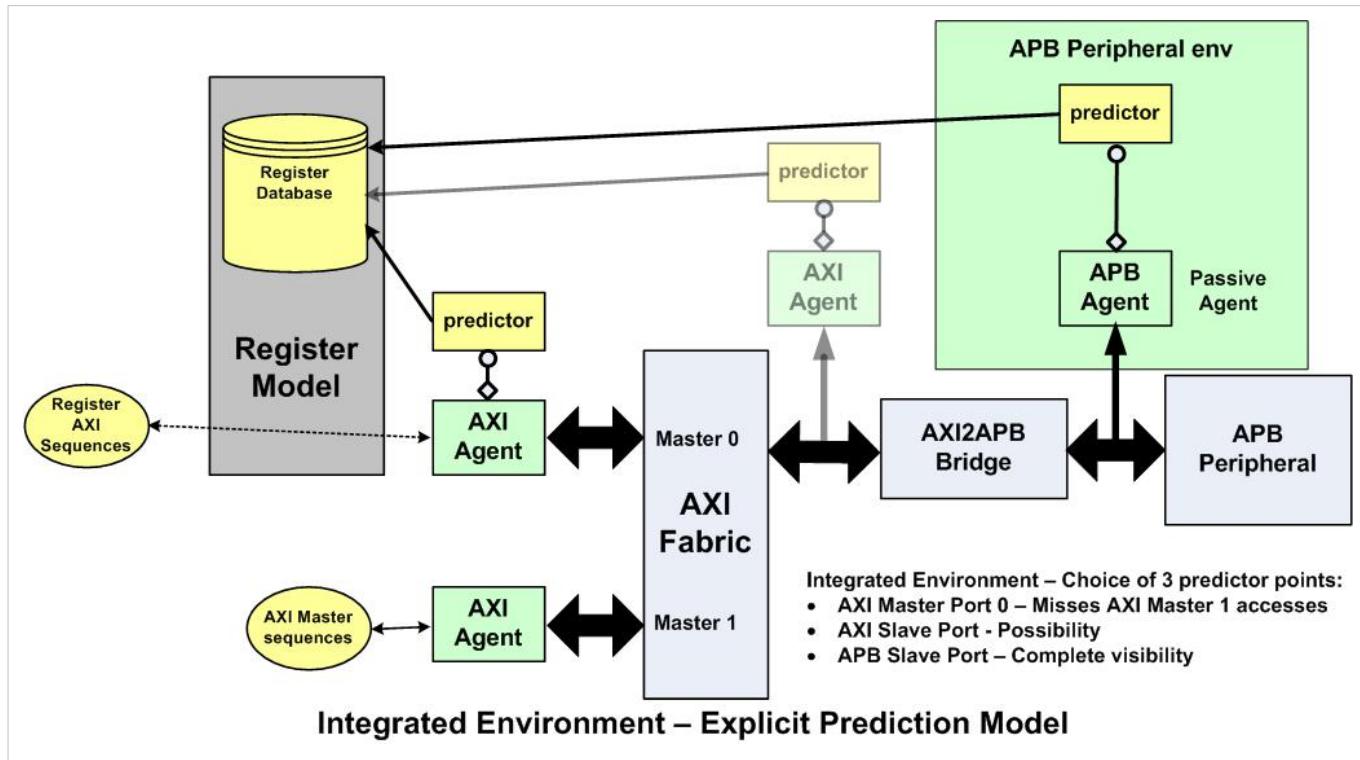
## Passive Prediction

In passive prediction the register model takes no active part in any accesses to the DUT, but it is kept up to date by the predictor when any front door register accesses take place.



## Vertically Integrated Environment Example

In the integrated environment example in the picture below, the APB peripheral is connected to an AXI bus interconnect fabric via a bus bridge. The peripheral can be accessed by either of the two bus masters, one of which runs sequences directly and the other uses the register model to generate accesses.



In this scenario there are a number of places where the predictor could be placed. If it is placed on the Master 0 AXI agent, then the register model is updated only as a result of accesses that it generates. If the predictor is placed on the slave AXI port agent, then the predictor will be able to update the register model on accesses from both AXI masters. However, if the predictor is placed on the APB bus it will be able to verify that the correct address mapping is being used for the APB peripheral and that bus transfers are behaving correctly end to end.

In this example, there could also be up to three address translations taking place (AXI Master 0 to AXI Slave, AXI Master 1 to AXI Slave, AXI Slave to APB) and the predictor used would need to use the right register model map when making the call to predict() a target registers mirrored value.

## Integrating a register model - Implementation Process

The recommended register model integration approach is to use explicit prediction, since this has a number of advantages, not least of which is that it will facilitate vertical reuse.

Based on the assumption that you have a register model, then you will need to follow the steps below in the order described to integrate a register model into your environment:

1. Understand which map in the register model correspond to which target bus agent
2. Check whether the bus agent has an UVM register model adapter class, otherwise you will have to implement one
3. Declare and build the register model in the test, passing a handle to it down the testbench hierarchy via configuration (or resources).
4. In each env that contains an active bus interface agent set up the bus layering

5. In each env that contains a bus agent set up a predictor

The register model you have will contain one or more maps which define the register address mapping for a specific bus interface. In most cases, block level testbenches will only require one map, but the register model has been designed to cope with the situation where a DUT has multiple bus interfaces which could quite easily occur in a multi-master SoC. Each map will need to have the adapter and predictor layers specified.

# Registers/Integration

The integration process for the register model involves constructing it and placing handles to it inside the relevant configuration objects, and then creating the adaption layers.

## Register Model Construction

The register model should be constructed in the test and a handle to it should be passed to the rest of the testbench hierarchy via configuration objects. In the case of a block level environment, a handle to the whole model will be passed. However, in the case of a cluster (sub-system) or a SoC environment, the overall register model will be an integration of register blocks for the sub-components and handles to sub-blocks of the register model will be relevant to different sub-environments.

For instance, in the case of a block level testbench for the SPI, the SPI env configuration object would contain a handle for the SPI register model which would be created and assigned in the test.

The following code is from the test base class for the SPI block level testbench, note that the register model is created using the factory, and then its build() method is called.

**Note:** The register model build() method will not be called automatically by the uvm\_component build() phase. The register model is **NOT** an uvm\_component, so its build() method has to be called explicitly to construct and configure the register model. If the register model build method is not called, the objects within the register model would not be constructed and initialised which means that subsequent testbench code that access the register model will fail, most likely with null handle errors.

```

//  

// From the SPI Test base class  

//  

// Build the env, create the env configuration  

// including any sub configurations and assigning virtual interfaces  

function void spi_test_base::build_phase(uvm_phase phase);  

    // env configuration  

    m_env_cfg = spi_env_config::type_id::create("m_env_cfg");  

    // Register model  

    // Enable all types of coverage available in the register model  

    uvm_reg::include_coverage("*", UVM_CVR_ALL);  

    // Create the register model:  

    spi_rm = spi_reg_block::type_id::create("spi_rm");  

    // Build and configure the register model  

    spi_rm.build();  

    // Assign a handle to the register model in the env config  

    m_env_cfg.spi_rm = spi_rm;  

    // etc.  

endfunction: build_phase

```

In the case where the SPI is part of a cluster, then the whole cluster register model (containing the SPI register model as a block) would be created in the test, and then the SPI env configuration object would be passed a handle to the SPI register block as a sub-component of the overall register model:

```

//  

// From the build method of the PSS test base class:  

//  

//  

// PSS - Peripheral sub-system with a hierarchical register model with the handle pss_reg  

//  

// SPI is a sub-block of the PSS which has a register model handle in its env config object  

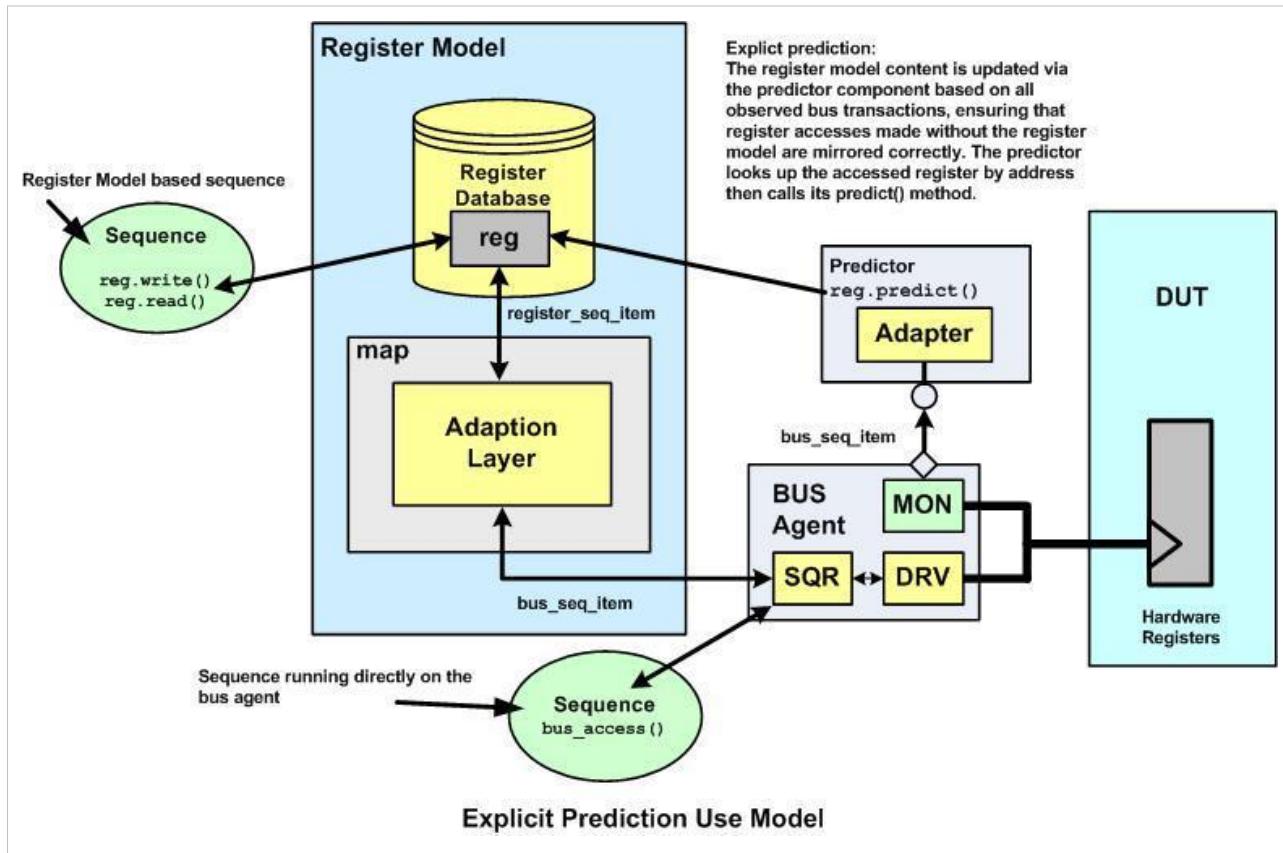
//  

m_spi_env_cfg.spi_rm = pss_reg.spi;

```

## Adaption Layer Implementation

There are two parts to the register adaption layer, the first part implements the sequence based stimulus layering and the second part implements the analysis based update of the register model using a predictor component.



## Register Sequence Adaption Layer

The register sequence layering adaption should be done during the UVM connect phase when the register model and the target agent components are known to have been built.

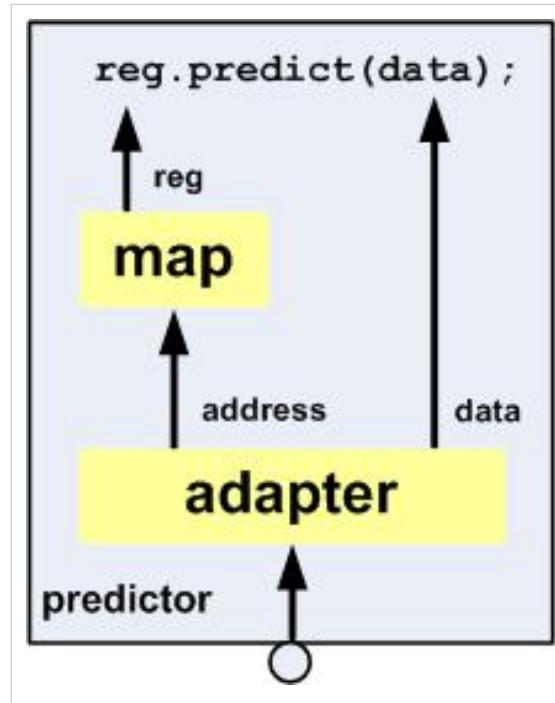
The register layering for each target bus interface agent supported by the register model should only be done once for each map. In a block level environment, this will be in the env, but in environments working at a higher level of integration this mapping should be done in the top level environment. In order to determine whether the particular env is at the top level the code should test whether the parent to its register block is null or not - if it is, then the model and therefore its env is at the top level.

In the following code from the SPI block level env connect method, the APB map from the SPI register block is layered onto the APB agent sequencer using the reg2apb adapter class using the set\_sequencer() method in APB\_map object:

```
//  
// From the SPI env  
//  
// Register layering adapter:  
reg2apb_adapter reg2apb;  
  
function void spi_env::connect_phase(uvm_phase phase);  
  if(m_cfg.m_apb_agent_cfg.active == UVM_ACTIVE) begin  
    reg2apb = reg2apb_adapter::type_id::create("reg2apb");  
    //  
    // Register sequencer layering part:  
    //  
    // Only set up register sequencer layering if the top level env  
    if(m_cfg.spi_rm.get_parent() == null) begin  
      m_cfg.spi_rm.APB_map.set_sequencer(m_apb_agent.m_sequencer, reg2apb);  
    end  
  end  
  
  //Predictor code shown below  
endfunction: connect_phase
```

## Register Prediction

By default, the register model uses a process called `explicit_predict` to update the register data base each time a read or write transaction that the model has generated completes. Explicit prediction requires the use of a `uvm_reg_predictor` component.



The `uvm_reg_predictor` component is derived from a `uvm_subscriber` and is parameterised with the type of the target bus analysis transaction. It contains handles for the register to target bus adapter and the register model map that is being used to interface to the bus agent sequencer. It uses the register adapter to convert the analysis transaction from the monitor to a register transaction, then it looks up the register by address in the register models bus specific map and modifies the contents of the appropriate register.

The `uvm_reg_predictor` component is part of the UVM library and does not need to be extended. However, to integrate it the following things need to be taken care of:

1. Declare the predictor using the target bus `sequence_item` as a class specialisation parameter
2. Create the predictor in the env `build()` method
3. In the `connect` method - set the predictor map to the target register model register map
4. In the `connect` method - set the predictor adapter to the target agent adapter class
5. In the `connect` method - connect the predictor analysis export to the target agent analysis port

A predictor should be included at every place where there is a bus monitor on the target bus. The code required is shown below and is from the second half of the SPI `connect` method code.

```

// 
// From the SPI env
// 
// Register predictor:
uvm_reg_predictor #(apb_seq_item) apb2reg_predictor;

```

```
function void spi_env::build_phase(uvm_phase phase);
  if(!uvm_config_db #(spi_env_config)::get(this, "", "spi_env_config", m_cfg)) begin
    `uvm_error("build_phase", "SPI env configuration object not found")
  end
  m_cfg = spi_env_config::get_config(this);
  if(m_cfg.has_apb_agent) begin
    set_config_object("m_apb_agent*", "apb_agent_config", m_cfg.m_apb_agent_cfg, 0);
    m_apb_agent = apb_agent::type_id::create("m_apb_agent", this);
    // Build the register model predictor
    apb2reg_predictor = uvm_reg_predictor #(apb_seq_item)::type_id::create("apb2reg_predictor", this);
  end
  //
  // etc
  //
endfunction:build_phase

function void spi_env::connect_phase(uvm_phase phase);
  //Adapter Created in code shown above

  // Register prediction part:
  //
  // Replacing implicit register model prediction with explicit prediction
  // based on APB bus activity observed by the APB agent monitor
  // Set the predictor map:
  apb2reg_predictor.map = m_cfg.spi_rm.APB_map;
  // Set the predictor adapter:
  apb2reg_predictor.adapter = reg2apb;
  // Connect the predictor to the bus agent monitor analysis port
  m_apb_agent.ap.connect(apb2reg_predictor.bus_in);

endfunction : connect_phase
```

The code excerpts shown come from the *spi\_tb/tests/spi\_test\_base.svh* file (register model construction) and the *spi\_tb/env/spi\_env.svh* file (adapter and predictor) from the example download:

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# Registers/RegisterModelOverview

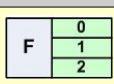
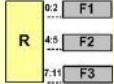
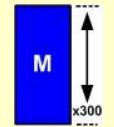
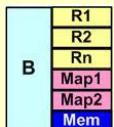
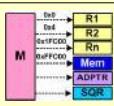
In order to be able to use the UVM register model effectively, it is important to have a mental model of how it is structured in order to be able to find your way around it.

The register model is implemented using five main building blocks - the register field; the register; the memory; the register block; and the register map. The register field models a collection of bits that are associated with a function within a register. A field will have a width and a bit offset position within the register. A field can have different access modes such as read/write, read only or write only. A register contains one or more fields. A register block corresponds to a hardware block and contains one or more registers. A register block also contains one or more register maps.

A memory region in the design is modelled by a `uvm_mem` which has a range, or size, and is contained within a register block and has an offset determined by a register map. A memory region is modelled as either read only, write only or read-write with all accesses using the full width of the data field. A `uvm_memory` does not contain fields.

The register map defines the address space offsets of one or more registers or memories in its parent block from the point of view of a specific bus interface. A group of registers may be accessible from another bus interface by a different set of address offsets and this can be modelled by using another address map within the parent block. The address map is also used to specify which bus agent is used when a register access takes place and which adapter is used to convert generic register transfers to/from target bus transfer sequence\_items.

This structure is illustrated in the following table:

Register base class	Purpose	Illustration
<code>uvm_reg_field</code>	Models functional groups of bits within a register	
<code>uvm_reg</code>	Collects fields spanning bit positions within the register	
<code>uvm_mem</code>	Models a range of memory locations	
<code>uvm_block</code>	Collects registers, memories, (sub-blocks) and maps	
<code>uvm_map</code>	Specifies register, memory and sub-block address offsets, target bus interface	

In the case of a block level register model, the register block will most likely only contain a collection of registers and a single address map. A cluster, or sub-system, level register model will have a register block which will contain other register model blocks for each of the sub-components in the cluster, and register maps for each of the bus interfaces. At this level, a register map can specify an offset address for a sub-block and the accesses to the registers within that sub-block will be adjusted to generate the right address within the cluster level map.

The register model structure is designed to be modular and reusable. At higher levels of integration, the cluster register model block can be a sub-block within, say, a SoC register block. Again, multiple register maps can be used to relocate the address offsets for the registers within the sub-blocks according to the bus interface used to access them.

## Register Model Data Types

The register model uses some specific data types in order to standardise the width of the address and data fields:

Type	Default width	`define	Description
uvm_reg_data_t	64 bits	`UVM_REG_DATA_WIDTH	Used for register data fields (uvm_reg, uvm_reg_field, uvm_mem)
uvm_reg_addr_t	64 bits	`UVM_REG_ADDR_WIDTH	Used for register address variables

Both of these types are based on the SystemVerilog bit type and are therefore 2 state. By default, they are 64 bits wide, but the width of the each type is determined by a `define which can be overloaded by specifying a new define on the compilation command line.

```

#
# To change the width of the register uvm_data_t
# using Questa
#
vlog +incdir+$ (UVM_HOME) /src +define+UVM_REG_DATA_WIDTH=24 $ (UVM_HOME) /src/uvm_pkg.sv

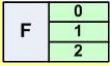
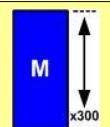
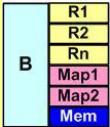
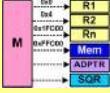
```

Since this is a compilation variable it requires that you recompile the UVM package and it also has a global effect impacting all register models, components and sequences which use these data types. Therefore, although it is possible to change the default width of this variable, it is not recommended since it could have potential side effects.

# Registers/ModelStructure

## Register Model Layer Overview

The UVM register model is built up in layers:

Layer	Description	Diagram
Fields	Bit(s) grouped according to function within a register	
Register	Collection of fields at different bit offset	
Memory	Represents a block of memory which extends over a specified range	
Block	Collection of registers (Hardware block level), or sub-blocks (Sub-system level) with one or more maps. May also include memories.	
Map	Named address map which locates the offset address of registers, memories or sub-blocks. Also defines the target sequencer for register accesses from the map.	

## Register Fields

The bottom layer is the field which corresponds to one or more bits within a register. Each field definition is an instantiation of the `uvm_reg_field` class. Fields are contained within an `uvm_reg` class and they are constructed and then configured using the `configure()` method:

```
//  
// uvm_field configure method prototype  
  
function void configure(uvm_reg parent, // The containing register  
                      int unsigned size, // How many bits wide  
                      int unsigned lsb_pos, // Bit offset within the register  
                      string access, // "RW", "RO", "WO" etc  
                      bit volatile, // Volatile if bit is updated by hardware  
                      uvm_reg_data_t reset, // The reset value  
                      bit has_reset, // Whether the bit is reset  
                      bit is_rand, // Whether the bit can be randomized  
                      bit individually_accessible); // i.e. Totally contained within a byte lane
```

How the `configure` method is used is shown in the register code example.

When the field is created, it takes its name from the string passed to its `create` method which by convention is the same as the name of its handle.

## Registers

Registers are modelled by extending the uvm\_reg class which is a container for field objects. The overall characteristics of the register are defined in its constructor method:

```
//  
// uvm_reg constructor prototype:  
  
//  
function new (string name="",           // Register name  
             int unsigned n_bits, // Register width in bits  
             int has_coverage); // Coverage model supported by the register
```

The register class contains a build method which is used to create and configure the fields. Note that this build method is not called by the UVM build phase, since the register is an uvm\_object rather than an uvm\_component. The following code example shows how the SPI master CTRL register model is put together.

```
//-----  
// ctrl  
//-----  
  
class ctrl extends uvm_reg;  
  `uvm_object_utils(ctrl)  
  
  rand uvm_reg_field acs;  
  rand uvm_reg_field ie;  
  rand uvm_reg_field lsb;  
  rand uvm_reg_field tx_neg;  
  rand uvm_reg_field rx_neg;  
  rand uvm_reg_field go_bsy;  
  uvm_reg_field reserved;  
  rand uvm_reg_field char_len;  
  
//-----  
// new  
//-----  
  
function new(string name = "ctrl");  
  super.new(name, 14, UVM_NO_COVERAGE);  
endfunction  
  
//-----  
// build  
//-----  
  
virtual function void build();  
  acs = uvm_reg_field::type_id::create("acs");  
  ie = uvm_reg_field::type_id::create("ie");  
  lsb = uvm_reg_field::type_id::create("lsb");  
  tx_neg = uvm_reg_field::type_id::create("tx_neg");
```

```

rx_neg = uvm_reg_field::type_id::create("rx_neg");
go_bsy = uvm_reg_field::type_id::create("go_bsy");
reserved = uvm_reg_field::type_id::create("reserved");
char_len = uvm_reg_field::type_id::create("char_len");

acs.configure(this, 1, 13, "RW", 0, 1'b0, 1, 1, 0);
ie.configure(this, 1, 12, "RW", 0, 1'b0, 1, 1, 0);
lsb.configure(this, 1, 11, "RW", 0, 1'b0, 1, 1, 0);
tx_neg.configure(this, 1, 10, "RW", 0, 1'b0, 1, 1, 0);
rx_neg.configure(this, 1, 9, "RW", 0, 1'b0, 1, 1, 0);
go_bsy.configure(this, 1, 8, "RW", 0, 1'b0, 1, 1, 0);
reserved.configure(this, 1, 7, "RO", 0, 1'b0, 1, 0, 0);
char_len.configure(this, 7, 0, "RW", 0, 7'b0000000, 1, 1, 0);

endfunction
endclass

```

When a register is added to a block it is created, causing its fields to be created and configured, and then it is configured before it is added to one or more reg\_maps to define its memory offset. The prototype for the register configure() method is as follows:

```

// 
// Register configure method prototype
// 

function void configure (uvm_reg_block blk_parent,           // The containing reg block
                        uvm_reg_file regfile_parent = null, // Optional, not used
                        string hdl_path = "");           // Used if HW register can be specified in one
                                              // hdl_path string

```

## Memories

Memories are modelled by extending the uvm\_mem class. The register model treats memories as regions, or memory address ranges where accesses can take place. Unlike registers, memory values are not stored because of the workstation memory overhead involved.

The range and access type of the memory is defined via its constructor:

```

// 
// uvm_mem constructor prototype:
// 

function new (string          name,           // Name of the memory model
              longint unsigned size,          // The address range
              int unsigned     n_bits,         // The width of the memory in bits
              string          access = "RW", // Access - one of "RW" or "RO"
              int             has_coverage = UVM_NO_COVERAGE); // Functional coverage

```

An example of a memory class implementation:

```

// Memory array 1 - Size 32'h2000;
class mem_1_model extends uvm_mem;

`uvm_object_utils(mem_1_model)

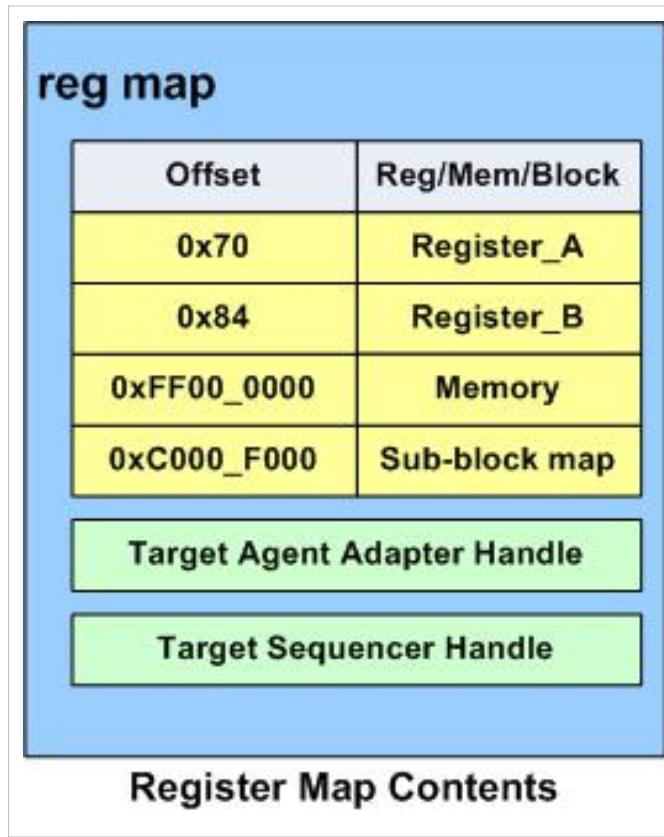
function new(string name = "mem_1_model");
    super.new(name, 32'h2000, 32, "RW", UVM_NO_COVERAGE);
endfunction

endclass: mem_1_model

```

## Register Maps

The purpose of the register map is two fold. The map provides information on the offset of the registers, memories and/or register blocks contained within it. The map is also used to identify which bus agent register based sequences will be executed on, however this part of the register maps functionality is set up when integrating the register model into an UVM testbench.



In order to add a register or a memory to a map, the `add_reg()` or `add_mem()` methods are used. The prototypes for these methods are very similar:

```

// 
// uvm_map add_reg method prototype:
// 
function void add_reg (uvm_reg rg,           // Register object handle

```

```

        uvm_reg_addr_t      offset,           // Register address offset
        string              rights = "RW",   // Register access policy
        bit                 unmapped=0,      // If true, register does not appear in the address map
                                         // and a frontdoor access needs to be defined
        uvm_reg_frontdoor  frontdoor=null); // Handle to register frontdoor access object
//
// uvm_map add_mem method prototype:
//
function void add_mem (uvm_mem      mem,           // Memory object handle
                      uvm_reg_addr_t offset,           // Memory address offset
                      string              rights = "RW",   // Memory access policy
                      bit                 unmapped=0,      // If true, memory is not in the address map
                                         // and a frontdoor access needs to be defined
                      uvm_reg_frontdoor  frontdoor=null); // Handle to memory frontdoor access object

```

There can be several register maps within a block, each one can specify a different address map and a different target bus agent.

## Register Blocks

The next level of hierarchy in the UVM register structure is the `uvm_reg_block`. This class can be used as a container for registers and memories at the block level, representing the registers at the hardware functional block level, or as a container for multiple blocks representing the registers in a hardware sub-system or a complete SoC organised as blocks. In order to define register and memory address offsets the block contains an address map object derived from `uvm_reg_map`. A register map has to be created within the register block using the `create_map` method:

```

//
// Prototype for the create_map method
//
function uvm_reg_map create_map(string name,           // Name of the map handle
                                 uvm_reg_addr_t base_addr, // The maps base address
                                 int unsigned n_bytes,    // Map access width in bytes
                                 uvm_endianness_e endian, // The endianess of the map
                                 bit byte_addressing=1); // Whether byte_addressing is supported

//
// Example:
//
AHB_map = create_map("AHB_map", 'h0, 4, UVM_LITTLE_ENDIAN);

```

### Note:

The `n_bytes` parameter is the word size (bus width) of the bus to which the map is associated. If a register's width exceeds the bus width, more than one bus access is needed to read and write that register over that bus. The `byte_addressing` argument affects how the address is incremented in these consecutive accesses. For example, if `n_bytes=4` and `byte_addressing=0`, then an access to a register that is 64-bits wide and at offset 0 will result in two bus accesses at addresses 0 and 1. With `byte_addressing=1`, that same access will result in two bus accesses at addresses 0

and 4.

In UVM 1.0, the default value for *byte\_addressing* was 0. In UVM 1.1 and later, the default for *byte\_addressing* is 1.

The first map to be created within a register block is assigned to the *default\_map* member of the register block.

The following code example is for the SPI master register block, this declares the register class handles for each of the registers in the SPI master, then the build method constructs and configures each of the registers before adding them to the APB\_map *reg\_map* at the appropriate offset address:

```
//-----
// spi_reg_block
//-----

class spi_reg_block extends uvm_reg_block;
  `uvm_object_utils(spi_reg_block)

  rand rxtx0 rxtx0_reg;
  rand rxtx1 rxtx1_reg;
  rand rxtx2 rxtx2_reg;
  rand rxtx3 rxtx3_reg;
  rand ctrl ctrl_reg;
  rand divider divider_reg;
  rand ss ss_reg;

  uvm_reg_map APB_map; // Block map

//-----
// new
//-----

function new(string name = "spi_reg_block");
  super.new(name, UVM_NO_COVERAGE);
endfunction

//-----
// build
//-----

virtual function void build();
  rxtx0_reg = rxtx0::type_id::create("rxtx0");
  rxtx0_reg.configure(this, null, "");
  rxtx0_reg.build();

  rxtx1_reg = rxtx1::type_id::create("rxtx1");
  rxtx1_reg.configure(this, null, "");
  rxtx1_reg.build();

  rxtx2_reg = rxtx2::type_id::create("rxtx2");
  rxtx2_reg.configure(this, null, "");
```

```

rxtx2_reg.build();

rxtx3_reg = rxtx3::type_id::create("rxtx3");
rxtx3_reg.configure(this, null, "");
rxtx3_reg.build();

ctrl_reg = ctrl::type_id::create("ctrl");
ctrl_reg.configure(this, null, "");
ctrl_reg.build();

divider_reg = divider::type_id::create("divider");
divider_reg.configure(this, null, "");
divider_reg.build();

ss_reg = ss::type_id::create("ss");
ss_reg.configure(this, null, "");
ss_reg.build();

// Map name, Offset, Number of bytes, Endianess
APB_map = create_map("APB_map", 'h0, 4, UVM_LITTLE_ENDIAN);

APB_map.add_reg(rxtx0_reg, 32'h00000000, "RW");
APB_map.add_reg(rxtx1_reg, 32'h00000004, "RW");
APB_map.add_reg(rxtx2_reg, 32'h00000008, "RW");
APB_map.add_reg(rxtx3_reg, 32'h0000000c, "RW");
APB_map.add_reg(ctrl_reg, 32'h00000010, "RW");
APB_map.add_reg(divider_reg, 32'h00000014, "RW");
APB_map.add_reg(ss_reg, 32'h00000018, "RW");

lock_model();
endfunction

endclass

```

Note that the final statement in the build method is the lock\_model() method. This is used to finalise the address mapping and to ensure that the model cannot be altered by another user.

The register block in the example can be used for block level verification, but if the SPI is integrated into a larger design, the SPI register block can be combined with other register blocks in an integration level block to create a new register model. The cluster block incorporates each sub-block and adds them to a new cluster level address map. This process can be repeated and a full SoC register map might contain several nested layers of register blocks. The following code example shows how this would be done for a sub-system containing the SPI master and a number of other peripheral blocks.

```
package pss_reg_pkg;
```

```

import uvm_pkg::*;
`include "uvm_macros.svh"

import spi_reg_pkg::*;
import gpio_reg_pkg::*;

class pss_reg_block extends uvm_reg_block;

`uvm_object_utils(pss_reg_block)

function new(string name = "pss_reg_block");
    super.new(name);
endfunction

rand spi_reg_block spi;
rand gpio_reg_block gpio;

function void build();
    AHB_map = create_map("AHB_map", 0, 4, UVM_LITTLE_ENDIAN);

    spi = spi_reg_block::type_id::create("spi");
    spi.configure(this);
    spi.build();
    AHB_map.add_submap(this.spi.default_map, 0);

    gpio = gpio_reg_block::type_id::create("gpio");
    gpio.configure(this);
    gpio.build();
    AHB_map.add_submap(this gpio.default_map, 32'h100);

    lock_model();
endfunction: build

endclass: pss_reg_block

endpackage: pss_reg_pkg

```

If the hardware register space can be accessed by more than one bus interface, then the block can contain multiple address maps to support alternative address maps. In the following example, two maps are created and have memories and registers added to them at different offsets:

```

// 
// Memory sub-system (mem_ss) register & memory block
// 

```

```
class mem_ss_reg_block extends uvm_reg_block;  
  
`uvm_object_utils(mem_ss_reg_block)  
  
function new(string name = "mem_ss_reg_block");  
    super.new(name, build_coverage(UVM_CVR_ADDR_MAP));  
endfunction  
  
// Mem array configuration registers  
rand mem_offset_reg mem_1_offset;  
rand mem_range_reg mem_1_range;  
rand mem_offset_reg mem_2_offset;  
rand mem_range_reg mem_2_range;  
rand mem_offset_reg mem_3_offset;  
rand mem_range_reg mem_3_range;  
rand mem_status_reg mem_status;  
  
// Memories  
rand mem_1_model mem_1;  
rand mem_2_model mem_2;  
rand mem_3_model mem_3;  
  
// Map  
uvm_reg_map AHB_map;  
uvm_reg_map AHB_2_map;  
  
function void build();  
    mem_1_offset = mem_offset_reg::type_id::create("mem_1_offset");  
    mem_1_offset.configure(this, null, "");  
    mem_1_offset.build();  
    mem_1_range = mem_range_reg::type_id::create("mem_1_range");  
    mem_1_range.configure(this, null, "");  
    mem_1_range.build();  
    mem_2_offset = mem_offset_reg::type_id::create("mem_2_offset");  
    mem_2_offset.configure(this, null, "");  
    mem_2_offset.build();  
    mem_2_range = mem_range_reg::type_id::create("mem_2_range");  
    mem_2_range.configure(this, null, "");  
    mem_2_range.build();  
    mem_3_offset = mem_offset_reg::type_id::create("mem_3_offset");  
    mem_3_offset.configure(this, null, "");  
    mem_3_offset.build();  
    mem_3_range = mem_range_reg::type_id::create("mem_3_range");  
    mem_3_range.configure(this, null, "");
```

```
mem_3_range.build();

mem_status = mem_status_reg::type_id::create("mem_status");
mem_status.configure(this, null, "");
mem_status.build();

mem_1 = mem_1_model::type_id::create("mem_1");
mem_1.configure(this, "");

mem_2 = mem_2_model::type_id::create("mem_2");
mem_2.configure(this, "");

mem_3 = mem_3_model::type_id::create("mem_3");
mem_3.configure(this, "");

// Create the maps

AHB_map = create_map("AHB_map", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
AHB_2_map = create_map("AHB_2_map", 'h0, 4, UVM_LITTLE_ENDIAN, 1);

// Add registers and memories to the AHB_map

AHB_map.add_reg(mem_1_offset, 32'h00000000, "RW");
AHB_map.add_reg(mem_1_range, 32'h00000004, "RW");
AHB_map.add_reg(mem_2_offset, 32'h00000008, "RW");
AHB_map.add_reg(mem_2_range, 32'h0000000c, "RW");
AHB_map.add_reg(mem_3_offset, 32'h00000010, "RW");
AHB_map.add_reg(mem_3_range, 32'h00000014, "RW");
AHB_map.add_reg(mem_status, 32'h00000018, "RO");

AHB_map.add_mem(mem_1, 32'hF000_0000, "RW");
AHB_map.add_mem(mem_2, 32'hA000_0000, "RW");
AHB_map.add_mem(mem_3, 32'h0001_0000, "RW");

// Add registers and memories to the AHB_2_map

AHB_2_map.add_reg(mem_1_offset, 32'h8000_0000, "RW");
AHB_2_map.add_reg(mem_1_range, 32'h8000_0004, "RW");
AHB_2_map.add_reg(mem_2_offset, 32'h8000_0008, "RW");
AHB_2_map.add_reg(mem_2_range, 32'h8000_000c, "RW");
AHB_2_map.add_reg(mem_3_offset, 32'h8000_0010, "RW");
AHB_2_map.add_reg(mem_3_range, 32'h8000_0014, "RW");
AHB_2_map.add_reg(mem_status, 32'h8000_0018, "RO");

AHB_2_map.add_mem(mem_1, 32'h7000_0000, "RW");
AHB_2_map.add_mem(mem_2, 32'h2000_0000, "RW");
AHB_2_map.add_mem(mem_3, 32'h0001_0000, "RW");

lock_model();

endfunction: build

endclass: mem_ss_reg_block
```

# Registers/QuirkyRegisters

## Introduction

Quirky registers are just like any other register described using the register base class except for one thing. They have special (quirky) behavior that either can't be described using the register base class, or is hard to describe using the register based class. The register base class can be used to describe the behavior of many different registers - for example clean-on -read (RC), write-one-to-set (W1S), write-zero-to-set (W0S). These built-in behaviors are set using attributes. Setting the attribute caused the built-in behavior. Built-in behaviors can be used for a majority of most register descriptions, but most verification environments have a small number of special registers with behavior that can't be described but the built-in attributes. These are quirky registers.

Examples of quirky registers include 'clear on the third read', 'read a register that is actually a collection of bits from 2 other registers'. These are registers that have very special behavior. Quirky registers are outside the register base class functionality and are most easily implemented by extending the base class functions or by adding callbacks.

The register base class library is very powerful and offers many ways to change behavior. The easiest implementations extend the underlying register or field class, and redefine certain virtual functions or tasks, like set(), or get(), or read(). In addition to replacing functions and tasks, callbacks can be added. A callback is called at specific times from within the base class functions. For example, the post\_predict() callback is called from the uvm\_reg\_field::predict() call. Adding a post\_predict() callback, allows the field value to be changed (predicted). The UVM user guide and reference guide have much more information on register function and task overloading as well as callback definition.

## Quirky registers built-in to the library

Some special behaviors are delivered as part of the library - uvm\_reg\_fifo.svh and uvm\_reg\_indirect.svh. These classes implement fifo behavior and indirect behavior respectively. Building a fifo register can be done by extending this built-in class - uvm\_reg\_fifo. (See examples/register/models/fifo\_reg/reg\_model.sv)

```
class fifo_reg extends uvm_reg_fifo;

    function new(string name = "fifo_reg");
        super.new(name, 8, 32, UVM_NO_COVERAGE);
    endfunction: new

    `uvm_object_utils(fifo_reg)

endclass
```

## Quirky register examples in the library

Some other special behaviors are captured as examples in the delivered kit. These special behaviors are not as general as fifo or indirect, and so are not included in the base class library itself. These examples are a rich source of information about how to use the register base class to model quirky registers. For example, a shared register that is accessible from two different interfaces (AHB and Wishbone) can be seen in examples/simple/registers/models/shared\_reg/reg\_B.sv.

If your quirky behavior is similar to one of these choices, then the recommendation is to review that code, and either reuse it, extend it, or enhance it.

## A Custom Quirky Register

If your quirky behavior does not match any of the choices, then you'll need to build your own new register functionality. The example below implements an ID register. An ID register returns an element from a list on each successive read. Each read returns the next item on the list. When the end of the list is reached, the first element is returned. When the ID register is written, the write data causes the list pointer to become the value written. For example, writing 2 to the ID register will cause the third item from the list to be returned on the next read.

### ID Register

A snapshot of some code that implements an ID register is below. (See the full example for the complete text).

```
static int id_register_pointer      = 0;
static int id_register_pointer_max = 10;
static int id_register_value[] =
{'ha0, 'ha1, 'ha2, 'ha3, 'ha4,
 'ha5, 'ha6, 'ha7, 'ha8, 'ha9};

static task rw(reg_rw rw);
  `uvm_info("DUT REQ", rw.convert2string(), UVM_INFO)
  case (rw.addr)
    'h000: begin
      // The ID Register.
      // On a read, grab the value and advance the pointer.
      // On a write, update the pointer.
      // If the pointer wraps past MAX, set it to zero.
      if (rw.read)
        rw.data = id_register_value[id_register_pointer++];
      else
        id_register_pointer = rw.data;

      if (id_register_pointer >= id_register_pointer_max)
        id_register_pointer = 0;
      `uvm_info("DUT RSP", rw.convert2string(), UVM_INFO)
    end
  ...

```

## ID Register Model

The ID register model is implemented below. The register itself is similar to a regular register, except the ID register uses a new kind of field - the id\_register\_field.

The ID register field implements the specific functionality of the ID register in this case.

```
// The ID Register.

// Just a register which has a special field - the
// ID Register field.

class id_register extends uvm_reg;
    id_register_field F1;

    function new(string name = "id_register");
        super.new(name, 8, UVM_NO_COVERAGE);
    endfunction: new

    virtual function void build();
        F1 = id_register_field::type_id::create("F1",
            get_full_name());
        F1.configure(this, 8, 0, "RW", 0, 8'ha0, 1, 0, 1);
        begin
            id_register_field_cbs cb = new();
            // Setting the name makes the messages prettier.
            cb.set_name("id_register_field_cbs");
            uvm_reg_field_cb::add(F1, cb);
        end
    endfunction: build

    `uvm_object_utils(id_register)

endclass : id_register
```

The id\_register builds the fields in the build() routine, just like any other register. In addition to building the field, a callback is created and registered with the field. The callback in this case implements the post\_predict() method, and will be called from the underlying predict() code in the register field class.

```
begin
    id_register_field_cbs cb = new();
    // Setting the name makes the messages prettier.
    cb.set_name("id_register_field_cbs");
    uvm_reg_field_cb::add(F1, cb);
end
```

## ID Register Model Field

The ID register field is a simple implementation of the functionality required. The function set() causes the pointer to be updated.

```
class id_register_field extends uvm_reg_field;
  `uvm_object_utils(id_register_field)

  int id_register_pointer = 0;
  int id_register_pointer_max = 10;
  int id_register_value[] =
    {'ha0, 'ha1, 'ha2, 'ha3, 'ha4,
     'ha5, 'ha6, 'ha7, 'ha8, 'ha9};

  function new(string name = "id_register_field");
    super.new(name);
  endfunction

  function uvm_reg_data_t get(string fname = "",
                             int      lineno = 0);
    set(id_register_pointer, fname, lineno);
    return super.get(fname, lineno);
  endfunction

  function void set(uvm_reg_data_t value,
                    string      fname = "",
                    int        lineno = 0);
    id_register_pointer = value;
    if (id_register_pointer >= id_register_pointer_max)
      id_register_pointer = 0;
    super.set( id_register_value[id_register_pointer],
               fname, lineno);
  endfunction
endclass
```

Calling get() causes the value to be returned, and the pointer to be adjusted properly. The set() and get() are overridden to implement the desired model functionality. The ID register field also contains the specific values that will be read back. Those values could be obtained externally, or could be set from some other location.

## ID Register Field Callback

The ID register field callback is of type `uvm_reg_cbs`, and implements `post_predict()` to achieve the desired functionality. The `post_predict()` function implementation is in terms of the `set()` and `get()` that exist in the field. Please refer to the reference guide for more information about defining and using register field callbacks.

```
typedef class id_register_field;

class id_register_field_cbs extends uvm_reg_cbs;
  `uvm_object_utils(id_register_field_cbs)

  virtual function void post_predict(input uvm_reg_field  fld,
                                     input uvm_reg_data_t previous,
                                     inout uvm_reg_data_t value,
                                     input uvm_predict_e  kind,
                                     input uvm_path_e     path,
                                     input uvm_reg_map    map);

    id_register_field my_field;
    $cast(my_field, fld);

    `uvm_info("DBG", $psprintf(
      "start: post_predict(value=%0x, pointer=%0d). (%s)",
      value, my_field.id_register_pointer, kind.name()),
      UVM_INFO)

    case (kind)
      UVM_PREDICT_READ:
        my_field.set(my_field.id_register_pointer+1);
      UVM_PREDICT_WRITE:
        my_field.set(value);
    endcase

    `uvm_info("DBG", $psprintf(
      " done: post_predict(fld.get()=%0x, pointer=%0d). (%s)",
      my_field.get(), my_field.id_register_pointer,
      kind.name()), UVM_INFO)
  endfunction
endclass
```

The callback created and registered from the register definition, but it is registered on the field.

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# Registers/ModelCoverage

## Controlling the build/inclusion of covergroups in the register model

Which covergroups get built within a register block object or a register object is determined by a local variable called `m_has_cover`. This variable is of type `uvm_coverage_model_e` and it should be initialised by a `build_coverage()` call within the constructor of the register object which assigns the value of the `include_coverage` resource to `m_has_cover`. Once this variable has been set up, the covergroups within the register model object should be constructed according to which coverage category they fit into.

The construction of the various covergroups would be based on the result of the `has_coverage()` call.

As each covergroup category is built, the `m_cover_on` variable needs to be set to enable coverage sampling on that set of covergroups, this needs to be done by calling the `set_coverage()` method.

## Controlling the sampling of covergroups in the register model

Depending on whether the covergroup needs to be sampled automatically on register access or as the result of an external `sample_values()` call, two different methods need to be implemented for each object. The `sample()` method is called automatically for each register and register block on each access. The `sample_values()` method is intended to be called from elsewhere in the testbench. In both cases, these methods test the `m_cover_on` bit field using the `has_coverage()` method to determine whether to sample a set of covergroups or not.

## The register model coverage control methods

The various methods used to control covergroup build and their effects are summarised here:

Method	Description
<b>Overall Control</b>	
<code>uvm_reg::include_coverage(uvm_coverage_model_e)</code>	Static method that sets up a resource with the key "include_coverage". Used to control which types of coverage are collected by the register model
<b>Build Control</b>	
<code>build_coverage(uvm_coverage_model_e)</code>	Used to set the local variable <code>m_has_cover</code> to the value stored in the resource database against the "include_coverage" key
<code>has_coverage(uvm_coverage_model_e)</code>	Returns true if the coverage type is enabled in the <code>m_has_cover</code> field
<code>add_coverage(uvm_coverage_model_e)</code>	Allows the coverage type(s) passed in the argument to be added to the <code>m_has_cover</code> field
<b>Sample Control</b>	
<code>set_coverage(uvm_coverage_model_e)</code>	Enables coverage sampling for the coverage type(s), sampling is not enabled by default
<code>get_coverage(uvm_coverage_model_e)</code>	Returns true if the coverage type(s) are enabled for sampling

*Note: That it is not possible to set an enable coverage field unless the corresponding build coverage field has been set.*

## An example

The following code comes from a register model implementation of a register model that incorporates functional coverage.

In the test, the overall register coverage model for the testbench has to be set using the `uvm_reg::include_coverage()` static method:

```
//
// Inside the test build method:
//
function void spi_test_base::build();
    uvm_reg::include_coverage(UVM_CVR_ALL); // All register coverage types enabled
//
//...
```

The first code excerpt is for a covergroup which is intended to be used at the block level to get read and write access coverage for a register block. The covergroup has been wrapped in a class included in the register model package, this makes it easier to work with.

```
//
// A covergroup (wrapped in a class) that is designed to get the
// register map read/write coverage at the block level
//
//
// This is a register access covergroup within a wrapper class
//
// This will need to be called by the block sample method
//
// One will be needed per map
//
class SPI_APB_reg_access_wrapper extends uvm_object;
    `uvm_object_utils(SPI_APB_reg_access_wrapper)

    covergroup ra_cov(string name) with function sample(uvm_reg_addr_t addr, bit is_read);
        option.per_instance = 1;
        option.name = name;

        // To be generated:
        //
        // Generic form for bins is:
        //
        // bins reg_name = {reg_addr};
        ADDR: coverpoint addr {
            bins rxtx0 = {'h0};
```

```

bins rxtx1 = {'h4};
bins rxtx2 = {'h8};
bins rxtx3 = {'hc};
bins ctrl = {'h10};
bins divider = {'h14};
bins ss = {'h18};
}

// Standard code - always the same
RW: coverpoint is_read {
    bins RD = {1};
    bins WR = {0};
}

ACCESS: cross ADDR, RW;

endgroup: ra_cov

function new(string name = "SPI_APB_reg_access_wrapper");
    ra_cov = new(name);
endfunction

function void sample(uvm_reg_addr_t offset, bit is_read);
    ra_cov.sample(offset, is_read);
endfunction: sample

endclass: SPI_APB_reg_access_wrapper

```

The second code excerpt is for the register block which includes the covergroup. The code has been stripped down to show the parts relevant to the handling of the coverage model.

In the constructor of the block there is a call to `build_coverage()`, this ANDs the coverage enum argument supplied with the overall testbench coverage setting, which has been set by the `uvm_reg::include_coverage()` method and sets up the `m_has_cover` field in the block with the result.

In the blocks `build()` method, the `has_coverage()` method is used to check whether the coverage model used by the access coverage block is enabled. If it is, the access covergroup is built and then the coverage sampling is enabled for its coverage model using `set_coverage()`.

In the `sample()` method, the `get_coverage()` method is used to check whether the coverage model is enabled for sampling, and then the covergroup is sampled, passing in the address and `is_read` arguments.

```

// 
// The relevant parts of the spi_rm register block:
// 
//-----
// spi_reg_block

```

```
-----  
class spi_reg_block extends uvm_reg_block;  
  `uvm_object_utils(spi_reg_block)  
  
  rand rxtx0 rxtx0_reg;  
  rand rxtx1 rxtx1_reg;  
  rand rxtx2 rxtx2_reg;  
  rand rxtx3 rxtx3_reg;  
  rand ctrl ctrl_reg;  
  rand divider divider_reg;  
  rand ss ss_reg;  
  
  uvm_reg_map APB_map; // Block map  
  
  // Wrapped APB register access covergroup  
  SPI_APB_reg_access_wrapper SPI_APB_access_cg;  
  
-----  
// new  
-----  
function new(string name = "spi_reg_block");  
  // build_coverage ANDs UVM_CVR_ADDR_MAP with the value set  
  // by the include_coverage() call in the test bench  
  // The result determines which coverage categories can be built by this  
  // region of the register model  
  super.new(name, build_coverage(UVM_CVR_ADDR_MAP));  
endfunction  
  
-----  
// build  
-----  
virtual function void build();  
  string s;  
  
  // Check that the address coverage is enabled  
  if(has_coverage(UVM_CVR_ADDR_MAP)) begin  
    SPI_APB_access_cg = SPI_APB_reg_access_wrapper::type_id::create("SPI_APB_access_cg");  
    // Enable sampling on address coverage  
    set_coverage(UVM_CVR_ADDR_MAP);  
  end  
  
  //  
  // Create, build and configure the registers ...  
  //
```

```
APB_map = create_map("APB_map", 'h0, 4, UVM_LITTLE_ENDIAN);

APB_map.add_reg(rxtx0_reg, 32'h00000000, "RW");
APB_map.add_reg(rxtx1_reg, 32'h00000004, "RW");
APB_map.add_reg(rxtx2_reg, 32'h00000008, "RW");
APB_map.add_reg(rxtx3_reg, 32'h0000000c, "RW");
APB_map.add_reg(ctrl_reg, 32'h00000010, "RW");
APB_map.add_reg(divider_reg, 32'h00000014, "RW");
APB_map.add_reg(ss_reg, 32'h00000018, "RW");
add_hdl_path("DUT", "RTL");

lock_model();
endfunction: build

// Automatically called when a block access occurs:
function void sample(uvm_reg_addr_t offset, bit is_read, uvm_reg_map map);
    // Check whether coverage sampling is enabled for address accesses:
    if(get_coverage(UVM_CVR_ADDR_MAP)) begin
        // Sample the covergroup if access is for the APB_map
        if(map.get_name() == "APB_map") begin
            SPI_APB_access_cg.sample(offset, is_read);
        end
    end
endfunction: sample

endclass: spi_reg_block
//
```

# Registers/BackdoorAccess

The UVM register model facilitates access to hardware registers in the DUT either through front door accesses or back door accesses. A front door access involves using a bus transfer cycle using the target bus agent, consequently it consumes time, taking at least a clock cycle to complete and so it models what will happen to the DUT in real life. A backdoor access uses the simulator database to directly access the register signals within the DUT, with write direction operations forcing the register signals to the specified value and read direction accesses returning the current value of the register signals. A backdoor access takes zero simulation time since it by-passes the normal bus protocol.

## Defining The Backdoor HDL Path

To use backdoor accesses with the UVM register model, the user has to specify the hardware, or hdl, path to the signals that a register model represents. To aid reuse and portability, the hdl path is specified in hierarchical sections. Therefore the top level block would specify a path to the top level of the DUT, the sub-system block would have a path from within the DUT to the sub-system, and a register would have a path specified from within the sub-system. The register level hdl path also has to specify which register bit(s) correspond to the target hdl signal.

As an example, in the SPI master testbench, the SPI master is instantiated as "DUT" in the top level testbench, so the hdl path to the register block (which corresponds to the SPI master) is set to "DUT". Then the control register bits within the SPI master RTL is collected together in a vectored reg called "ctrl", so the hdl path to the control register is DUT.ctrl. The hdl path slice for the control register is set to "ctrl" in the build method of the SPI register block.

```
function void spi_reg_block::build();
    //
    // ....
    //
    ctrl_reg = ctrl::type_id::create("ctrl");
    ctrl_reg.build();
    // Can add the hdl path as last argument to configure but only if the whole register
    // content is contained within the hdl path
    ctrl_reg.configure(this, null, "");
    // Add the ctrl hdl_path starting at bit 0, hardware target is 14 bits wide
    ctrl_reg.add_hdl_path_slice("ctrl", 0, 14);
    //
    // ....
    //
    // Assign DUT to the hdl path
    add_hdl_path("DUT", "RTL");
    lock_model();
endfunction: build
```

## Tradeoffs Between Front And Backdoor Accesses

Backdoor accesses should be used carefully. The following table summarises the key functional differences between the two access modes:

Backdoor Access	Frontdoor Access
Take zero simulation time	Use a bus transaction which will take at least one RTL clock cycle
Write direction accesses force the HW register bits to the specified value	Write direction accesses do a normal HW write
Read direction accesses return the current value of the HW register bits	Read direction accesses to a normal HW read, data is returned using the HW data path
In the UVM register model, backdoor accesses are always auto-predicted - the mirrored value reflects the HW value	Frontdoor accesses are predicted based on what the bus monitor observes
Only the register bits accessed are affected, side effects may occur when time advances depending on HW implementation	Side effects are modelled correctly
By-passes normal HW	Simulates real timing and event sequences, catches errors due to unknown interactions

Backdoor access can be a useful and powerful technique and some valid use models include:

- Configuration or re-configuration of a DUT - Putting it into a non-reset random state before configuring specific registers via the front door
- Adding an extra level of debug when checking data paths - Using a backdoor peek after a front door write cycle and before a front door read cycle can quickly determine whether the write and read data path is responsible for any errors
- Checking a buffer before it is read by front door accesses - Traps an error earlier, especially useful if the read process does a data transform, or has side-effects

Some invalid use models include:

- Use as an accelerator for register accesses - May be justified if there are other test cases that thoroughly verify the register interface
- Checking the DUT against itself- A potential pitfall whereby the DUT behaviour is taken as correct rather than the specified behaviour

## Potential Simulator Optimisation Issues

For the backdoor accesses to work, the simulator database VPI access routines need to be able to find the hdl signals. Most simulators optimise away the detailed structural information needed to do this to improve performance. Therefore to be able to use the backdoor access mechanism you will have to turn off these optimisations, at least for the signals that you would like to access via the backdoor.

For Questa the way to do this is to compile the design with the **vlog +acc** switch with the r,n and b options selected (**vlog +acc=rnb**). This switch can also be used for a specific hdl signal, or region to minimise the area of the design that is not going to be fully optimised. See the Questa user documentation for more information.

```
# Compile my design so that all of it has:
# registers (r)
# nets (n)
# vector bits (b)
# visible so that backdoor access will work correctly
```

```

vlog my_design +acc=rnb

# Compile my design so that only the f field in the r register in the b block is visible
# for backdoor access
vlog my_design +acc=rnb+/tb/dut/b/r/f

# Other paths can be added with extra +'s

```

## Backdoor Access For RTL And Gate Level

The structure of RTL and a gate level netlist is quite different and the same hdl path will not hold in both cases. To address this issue, the register model supports the specification of more than one hdl path. The backdoor hdl\_path can be set up for both RTL and Gate level, since the hdl path methods accept a string argument to group the hdl path definitions into different sets. By default the "RTL" hdl path is used for all hdl\_path definitions, but this default can be changed to "GATES" (or anything else) to define hdl\_path segments for a gate level netlist. The following code shows how the original example can be extended to add in a "GATES" set of hdl\_paths to correspond to a gate level netlist

```

// Adding hdl paths for the gate level of abstraction
function void spi_reg_block::build();
    //
    // ....
    //
    ctrl_reg = ctrl::type_id::create("ctrl");
    ctrl_reg.build();
    // Can add the hdl path as last argument to configure but only if the whole register
    // content is contained within the hdl path
    ctrl_reg.configure(this, null, "");
    // Add the ctrl hdl_path starting at bit 0, hardware target is 14 bits wide
    ctrl_reg.add_hdl_path_slice("ctrl", 0, 14); // "RTL" by default
    ctrl_reg.add_hdl_path_slice("ctrl_dff.q", 0, 14, "GATES"); // Gate level spec
    //
    // ....
    //
    // Assign DUT to the hdl path for both abstractions
    add_hdl_path("DUT", "RTL");
    add_hdl_path("DUT", "GATES");
    lock_model();
endfunction: build

```

## Registers/Generation

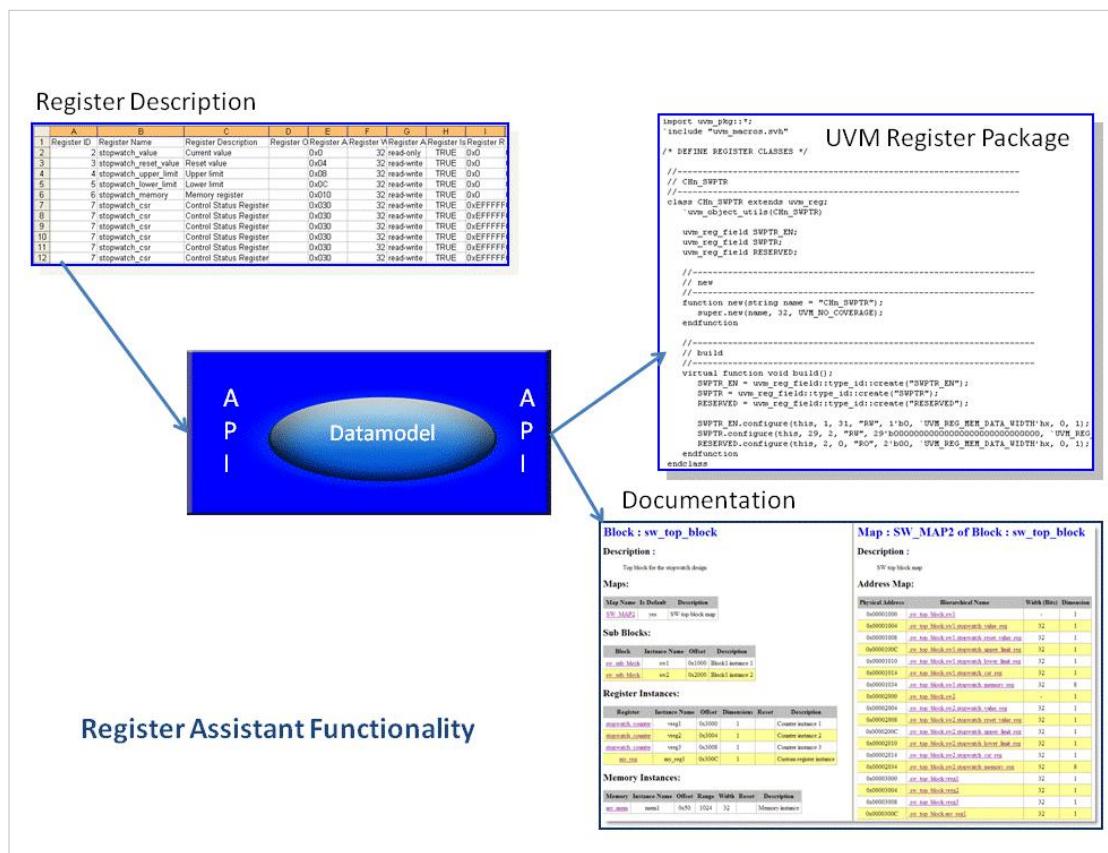
A register model can be written by hand, following the pattern given for the SPI master example. However, with more than a few registers this can become a big task and is always a potential source of errors. There are a number of other reasons why using a generator is helpful:

- It allows a common register specification to be used by the hardware, software and verification engineering teams
  - The register model can be generated efficiently without errors
  - The register model can be re-generated whenever there is a change in the register definition
  - Multiple format register definitions for different design blocks can be merged together into an overall register description

There are a number of register generators available commercially, including Mentor Graphics' Register Assistant.

As an option to Mentor Graphic's Certe Testbench Studio, Register Assistant reads in register descriptions and generates UVM register models and associated register documentation. Register descriptions can be read from spreadsheet (CSV), IP-XACT, and XML format inputs or via API commands within a script. If there are any changes to the register descriptions, the register package and documentation can be automatically updated. Register Assistant has an open datamodel that allows users to add their own readers, writers, or input checks. This capability combined with the API allows every aspect of the tool to be customized.

For more information about Certe Testbench Studio with Register Assistant, please see <http://www.mentor.com/certe>



# Registers/StimulusAbstraction

---

## Stimulus Abstraction

Stimulus that accesses memory mapped registers stimulus should be made as abstract as possible. The reasons for this are that it:

- Makes it easier for the implementer to write
- Makes it easier for users to understand
- Provides protection against changes in the register map during the development cycle
- Makes the stimulus easier to reuse

Of course, it is possible to write stimulus that does register reads and writes directly via bus agent sequence items with hard coded addresses and values - for instance `read(32'h1000_f104);` or `write(32'h1000_f108, 32'h05);` - but this stimulus would have to be re-written if the base address of the DUT changed and has to be decoded using the register specification during code maintenance.

The register model contains the information that links the register names to their addresses, and the register fields to their bit positions within the register. This means the register model makes it easier to write stimulus that is at a more meaningful level of abstraction - for instance `read(SPI.ctrl);` or `write(SPI.rxtx0, 32'h0000_5467);`.

The register model allows users to access registers and fields by name. For instance, if you have a SPI register model with the handle `spi_rm`, and you want to access the control register, `ctrl`, then the path to it is `spi_rm.ctrl`. If you want to access the `go_bsy` field within the control register then the path to it is `spi_rm.ctrl.go_bsy`.

Since the register model is portable, and can be quickly regenerated if there is a change in the specification of the register map, using the model allows the stimulus code to require minimal maintenance, once it is working.

The register model is integrated with the bus agents in the UVM testbench. What this means to the stimulus writer is that he uses register model methods to initiate transfers to/from the registers over the bus interface rather than using sequences which generate target bus agent sequence items. For the stimulus writer this reduces the amount of learning that is needed in order to become productive.

## UVM Register Data Value Tracking

The register model has its own database which is intended to represent the state of the hardware registers. For each register there is a mirrored value and a desired value. The desired value represents a state that the register model is going to use to update the hardware, but has not done so. In other words, the desired value allows the user to setup individual register fields before doing a write transfer. The mirrored value represents the current known state of the hardware register. The mirrored value is updated at the end of front bus read and write cycles either based on the data value seen by the register model (auto-prediction) or based on bus traffic observed by a monitor and sent to predictor that updates the register model content (recommended approach for integrating the register model). Backdoor accesses update the register model automatically. The mirrored value can become out of date over time if any of the bits within it are volatile, in other words, they are changed by hardware events rather than by being programmed.

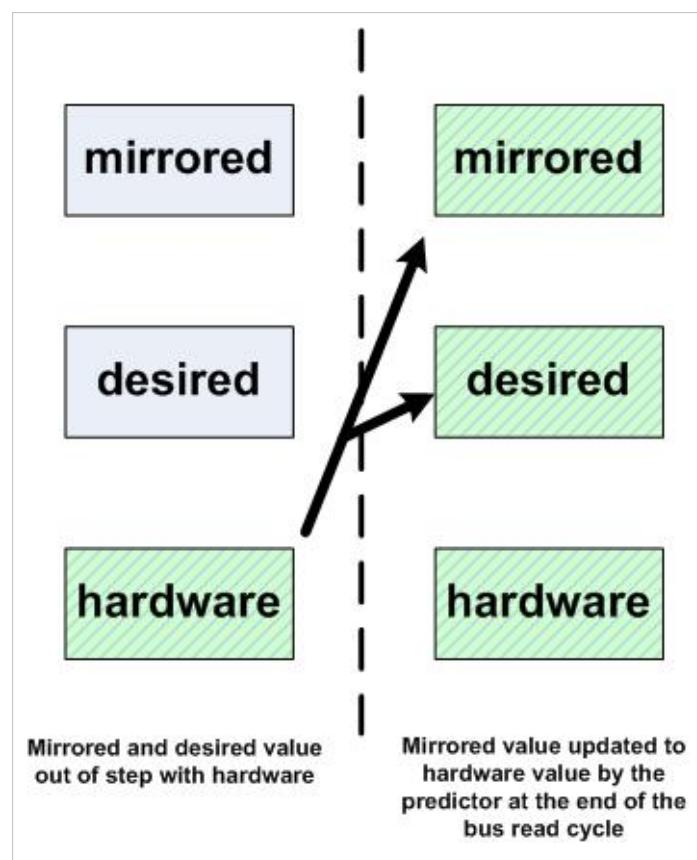
## UVM Register Access Methods

The register model has a number of methods which can be used to read and write registers in the DUT. These methods use the desired and mirrored values to keep in step with the actual hardware register contents.

The register model can either use front door or back door access to hardware registers. Front door access involves using the bus interface via an agent and simulates real life bus transfers with their associated timing and event relationships. Back door access uses simulator data base access routines to return the current value of the hardware register bits, or to force them to a particular value. Back door accesses happen in zero simulation time.

### read and write

The read() method returns the value of the hardware register. When using front door accesses, calling the read() method results in a bus transfer and the desired and mirrored values of the register model are updated by the bus predictor on completion of the read cycle.



```

// 
// read task prototype
// 

task read(output uvm_status_e      status,
          output uvm_reg_data_t   value,
          input  uvm_path_e       path = UVM_DEFAULT_PATH,
          input  uvm_reg_map      map = null,
          input  uvm_sequence_base parent = null,
          input  int               prior = -1,

```

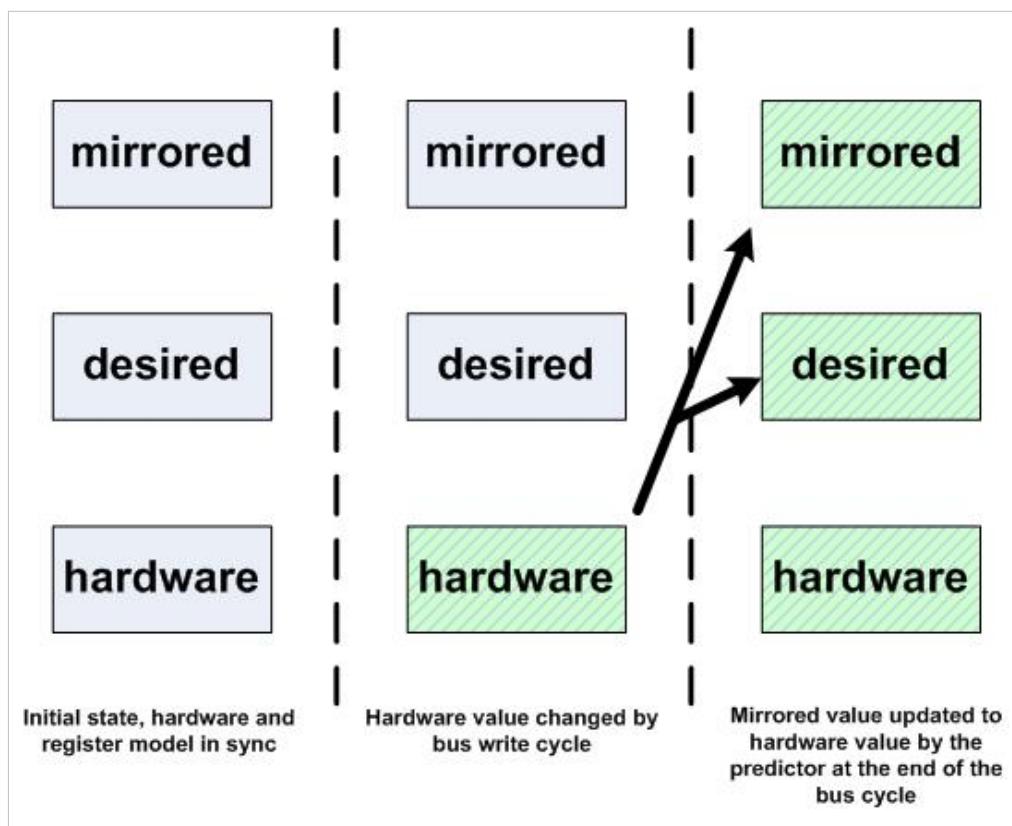
```

    input  uvm_object          extension = null,
    input  string               fname = "",
    input  int                  lineno = 0);

//
// Example - from within a sequence
//
// Note use of positional and named arguments
//
spi_rm.ctrl.read(status, read_data, .parent(this));

```

The write() method writes a specified value to the target hardware register. For front door accesses the mirrored and desired values are updated by the bus predictor on completion of the write cycle.



```

//
// write task prototype
//

task write(output uvm_status_e      status,
           input  uvm_reg_data_e  value,
           input  uvm_path_e      path = UVM_DEFAULT_PATH,
           input  uvm_reg_map     map = null,
           input  uvm_sequence_base parent = null,
           input  int              prior = -1,
           input  uvm_object       extension = null,

```

```

        input  string      fname = "",
        input  int       lineno = 0);

//
// Example - from within a sequence
//
// Note use of positional and named arguments
//
spi_rm.ctrl.write(status, write_data, .parent(this));

```

Although the read and write methods can be used at the register and field level, they should only be used at the register level to get predictable and reusable results. Field level reads and writes can only work if the field takes up and fills the whole of a byte lane when the target bus supports byte level access. Whereas this might work with register stimulus written with one bus protocol in mind, if the hardware block is integrated into a sub-system which uses a bus protocol that does not support byte enables, then the stimulus may no longer work.

The read() and write() access methods can also be used for back door accesses, and these complete and update the mirror value immediately.

### set and get

The set() and get() methods operate on the register model and do not result in any accesses to the hardware.

A call to *set* will assign an internal “desired value” for a register or field in the register model. This internal value will be a function of the value argument supplied to *set*, the current mirrored value, and the access policy. Calling *get* after a *set* will return the calculated desired value, which may or may not be the value you provided in the call to *set*.

Some examples:

- For RW access policies, the calculated desired value is always the value argument provided to *set*—the current mirrored value is irrelevant.
- For W1C, the desired value is the bitwise OR of the value argument and the current mirror value.
- For RO, the desired value is always the current mirror value, i.e. the ‘set’ has no effect whatsoever.
- For WC, the desired value is always 0, regardless of the value argument or current mirrored value.

If a *set* does not result in a desired value that is different than the current mirrored value, that field will not require an update. If all fields in a given register do not require an update, the register as a whole will not require an update, and therefore no bus activity will occur for that register upon a call to *update*.

It’s worth noting that multiple ‘sets’ to a register or field are cumulative until the *update* call. Normally, a register or field is set once before a call to *update*. But in cases where there are multiple sets, the register model uses the access policies to update the actual desired value each time. When *update* is called, the final desired value is applied on the bus, if necessary.

For example, for a 16-bit W1T (write 1 to toggle) register:

```

my_reg.set( 16'hFFFF ); // desired is now ~mirrored
my_reg.set( 16'hFFFF ); // desired is now what mirrored was originally
my_reg.update(...);

```

This would produce no bus activity because the calculated desired value is the the mirror value toggled twice, i.e. the cumulative result is the same as the current mirrored value, and so no update to the DUT is needed.

The following table summarizes the behavior of a register set/get operation based on each field's access mode, set value, and current mirror value.

- For register-level set/update operations, the table is applied to each field's access\_mode independently.
- A bus operation is necessary if any of the field's calculated desired values are different than their current mirror value.
- If multiple sets occur before an update, the mirror value column is the previous calculated desired value, D

### UVM Register set/update behavior vs access mode

Access Mode	Set value (A)	Mirror 'value (B)'	Desired 'value '(D)	Bus Access?	Value seen on Bus during update()
RO, RC, RS	x	x	no chg	N	-
RW, WRC, WRS, WO	A	B	A	If (B!=A)	D
WC, WCRS, WOC	x	B	0	If (B!=0)	D
WS, WSRC, WOS	x	B	all 1s	if (B!=1s)	D
W1C, W1CRS	A	B	$\sim A \& B$	If (B != D)	$\sim D$
W1S, W1SRC	A	B	$A \mid B$	If (B != D)	D
W1T	A	B	$A \wedge B$	If (B != D)	$D \wedge B$
W0C, W0CRS	A	B	$A \& B$	If (B != D)	D
W0S, W0SRC	A	B	$\sim A \mid B$	If (B != D)	$\sim D$
W0T	A	B	$\sim A \wedge B$	If (B != D)	$\sim(D \wedge B)$
W1,WO1	A	B	first ? A : B Note	If (B != D)	D

Because sets are cumulative until an *update* operation, the Mirror Value column can represent the previously calculated desired value from the last *set* operation. For W1 and WO1 modes, *first* is 1 until the first predicted write after a hard reset.

Examples:

The get() method returns the calculated desired value of a register or a field.

```

//  

// get function prototype  

//  

function uvm_reg_data_t get(string fname = "",  

                           int    lineno = 0);  

//  

// Examples - from within a sequence  

//  

uvm_reg_data_t ctrl_value;  

uvm_reg_data_t char_len_value;  

// Register level get:  

ctrl_value = spi_rm.ctrl.get();
```

```
// Field level get (char_len is a field within the ctrl reg):  
char_len_value = spi_rm.ctrl.char_len.get();
```

The set() method is used to setup the desired value of a register or a field prior to a write to the hardware using the update() method.

```
//  
// set function prototype  
//  
function void set(uvm_reg_data_t value,  
                  string      fname = "",  
                  int        lineno = 0);  
  
//  
// Examples - from within a sequence  
//  
uvm_reg_data_t ctrl_value;  
uvm_reg_data_t char_len_value;  
  
// Register level set:  
spi_rm.ctrl.set(ctrl_value);  
  
// Field level set (char_len is a field within the ctrl reg):  
spi_rm.ctrl.char_len.set(ctrl_value);
```

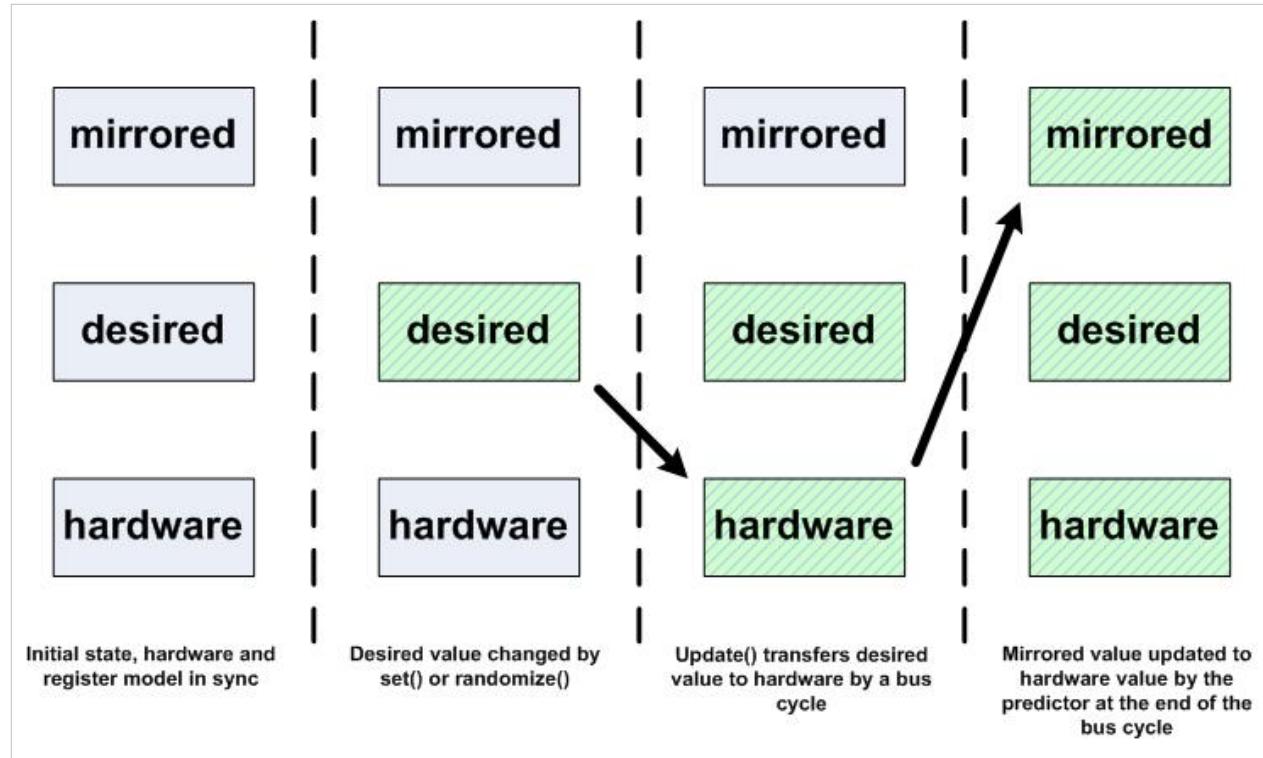
### **get\_mirrored\_value**

The get\_mirrored\_value() method is used to access the current mirrored value that is stored in the register model, like the set and get access methods it is an internal access method and does not cause any front or backdoor. It is intended to be used in scenarios where the current mirrored value of the register or field state is required without incurring the overhead of a register access.

Note that the get\_mirrored\_value() method was added in UVM 1.1a and was not present in UVM 1.1.

### update

If there is a difference in value between the desired and the mirrored register values, the update() method will initiate a write to a register. The update() can be called at the register level which will result in a single register write, or it can be called at the block level in which case it could result in several register writes. The mirrored value for the register would be set to the updated value by the predictor at the completion of the bus write cycle.



```

//  

// Prototype for the update task  

//  

task update(output uvm_status_e      status,  

            input  uvm_path_e        path = UVM_DEFAULT_PATH,  

            input  uvm_sequence_base parent = null,  

            input  int                prior = -1,  

            input  uvm_object         extension = null,  

            input  string             fname = "",  

            input  int                lineno = 0);  

//  

// Examples:  

//  

// Block level:  

spi_rm.update(status);  

//  

// Register level:  

spi_rm.ctrl.update(status);

```

Block level updates will always follow the same register access order. The update process indexes the register array in its database and the order used is dependent on the order that registers were created and added to the array by the register model. If randomized or variations in register access ordering are required then you should use individual register updates with your own ordering, perhaps using an array of register handles.

If multiple stimulus streams are active and using update() at the block level, then there is a chance that registers will be updated more than once, since multiple block level update() calls would be made.

### peek and poke

The peek() and poke() methods are backdoor access methods which can be used at the field and register level. The peek() method does a direct read of the hardware signal state and the poke() method forces the hardware signal state to match the data value. In both cases, the desired and mirrored values in the register model are updated automatically.

```
//  
// peek task prototype  
//  
task peek(output uvm_status_e      status,  
          output uvm_reg_data_t    value,  
          input  string           kind = "",  
          input  uvm_sequence_base parent = null,  
          input  uvm_object        extension = null,  
          input  string           fname = "",  
          input  int              lineno = 0);  
  
//  
// poke task prototype  
//  
task poke(output uvm_status_e      status,  
          input  uvm_reg_data_t    value,  
          input  string           kind = "",  
          input  uvm_sequence_base parent = null,  
          input  uvm_object        extension = null,  
          input  string           fname = "",  
          input  int              lineno = 0);  
  
//  
// Examples - from within a sequence  
//  
uvm_reg_data_t ctrl_value;  
uvm_reg_data_t char_len_value;  
  
// Register level peek:  
ctrl_value = spi_rm.ctrl.peek(status, ctrl_value, .parent(this));  
  
// Field level peek (char_len is a field within the ctrl reg):  
spi_rm.ctrl.char_len.peek(status, char_len_value, .parent(this));
```

```

// Register level poke:
spi_rm.ctrl.poke(status, ctrl_value, .parent(this));

// Field level poke:
spi_rm.ctrl.char_len.poke(status, char_len_value, .parent(this));

```

### randomize

Strictly speaking, this randomize() is not a register model method since the register model is based on SystemVerilog class objects. Depending on whether the registers and the register fields have been defined as rand or not, they can be randomized with or without constraints. The register model uses the post\_randomize() method to modify the desired register or field value. Subsequently, the hardware register can be written with the result of the randomization using the update() method.

The randomize() method can be called at the register model, block, register or field level.

### mirror

The mirror() method initiates a hardware read or peek access but does not return the hardware data value. A frontdoor read bus level operation results in the predictor updating the mirrored value, whereas a backdoor peek automatically updates the mirrored value. There is an option to check the value read back from the hardware against the original mirrored value.

The mirror() method can be called at the field, register or block level. In practice, it should only be used at the register or block level for front door accesses since field level read access may not fit the characteristics of the target bus protocol. A block level mirror() call will result in read/peek accesses to all of the registers within the block and any sub-blocks.

```

//
// mirror task prototype:
//

task mirror(output uvm_status_e      status,
            input  uvm_check_e      check = UVM_NO_CHECK,
            input  uvm_path_e       path  = UVM_DEFAULT_PATH,
            input  uvm_sequence_base parent = null,
            input  int              prior = -1,
            input  uvm_object        extension = null,
            input  string            fname = "",
            input  int              lineno = 0);

//
// Examples:
//
spi_rm.ctrl.mirror(status, UVM_CHECK); // Check the contents of the ctrl register
//
spi_rm.mirror(status, .path(UVM_BACKDOOR); // Mirror the contents of spi_rm block via the backdoor

```

**reset**

The reset() method sets the register desired and mirrored values to the pre-defined register reset value. This method should be called when a hardware reset is observed to align the register model with the hardware. The reset() method is an internal register model method and does not cause any bus cycles to take place.

The reset() method can be called at the block, register or field level.

```
//  
// reset function prototype:  
//  
function void reset(string kind = "HARD");  
//  
// Examples:  
//  
spi_rm.reset(); // Block level reset  
//  
spi_rm.ctrl.reset(); // Register level reset  
//  
spi_rm.ctrl.char_len.reset(); // Field level reset
```

**get\_reset**

The get\_reset() method returns the pre-defined reset value for the register or the field. It is normally used in conjunction with a read() or a mirror() to check that a register has been reset correctly.

```
//  
// get_reset function prototype:  
//  
function uvm_reg_data_t get_reset(string kind = "HARD");  
//  
// Examples:  
//  
uvm_reg_data_t ctrl_value;  
uvm_reg_data_t char_len_value;  
  
ctrl_value = spi_rm.ctrl.get_reset(); // Register level  
char_len_value = spi_rm.ctrl.char_len.get_reset(); // Field level
```

## UVM Register Access Method Arguments

Some of the register access methods contain a large number of arguments. Most of these have a default value so the user does not have to specify all of them when calling a method. The arguments required are more or less the same for those calls that require them and the following table summarises their purpose:

Argument	Type	Default Value	Purpose
status	uvm_status_e	None, must be populated with an argument	To return the status of the method call - can be UVM_IS_OK, UVM_NOT_OK, UVM_IS_X
value	uvm_reg_data_t	None	To pass a data value, an output in the read direction, an input in the write direction
path	uvm_path_e	UVM_DEFAULT_PATH	To specify whether a front or back door access is to be used - can be UVM_FRONTDOOR, UVM_BACKDOOR, UVM_PREDICT, UVM_DEFAULT_PATH
parent	uvm_sequence_base	null	To specify which sequence should be used when communicating with a sequencer
map	uvm_reg_map	null	To specify which register model map to use to make the access
prior	int	-1	To specify the priority of the sequence item on the target sequencer
extension	uvm_object	null	Allows an object to be passed in order to extend the call
fname	string	""	Used by reporting to tie method call to a file name
lineno	int	0	Used by reporting to tie method call to a line number
kind	string	"HARD"	Used to denote the type of reset

## UVM Register Access Method Summary

The following table summarises the various register access methods and the level at which they can be used for back door and front door accesses.

Method	Front door access			Back door access			Comment
	Block	Register	Field	Block	Register	Field	
read()	No	Yes	Not recommended	No	Yes	Yes	
write()	No	Yes	Not recommended	No	Yes	Yes	
get()	No	Yes	Yes	No	Yes	Yes	Internal Method
set()	No	Yes	Yes	No	Yes	Yes	Internal Method
peek()	No	Yes	Yes	No	Yes	Yes	
poke()	No	Yes	Yes	No	Yes	Yes	
randomize()	Yes	Yes	Yes	Yes	Yes	Yes	SV Class randomize()
update()	Yes	Yes	No	Yes	Yes	No	
mirror()	Yes	Yes	No	Yes	Yes	No	
reset()	Yes	Yes	Yes	Yes	Yes	Yes	Internal Method
get_reset()	No	Yes	Yes	No	Yes	Yes	Internal Method

## UVM Register Access Method Examples

To see examples of how to use the various register model access methods from sequences, please go to the SPI register based sequences example page.

# Registers/MemoryStimulus

## Memory Model Overview

The UVM register model also supports memory access. Memory regions within a DUT are represented by a memory models which have a configured width and range and are placed at an offset defined in a register map. The memory model is defined as having either a read-write (RW), a read-only (RO - ROM), or a write-only (WO) access type.

Unlike the register model, the memory model does not store state, it simply provides an access layer to the memory. The reasoning behind this is that storing the memory content would mean incurring a severe overhead in simulation and that the DUT hardware memory regions are already implemented using models which offer alternative verification capabilities. The memory model supports front door accesses through a bus agent, or backdoor accesses with direct access to the memory model content.

## Memory Model Access Methods

The memory model supports 4 types of access methods:

- read
- write
- burst\_read
- burst\_write

### Memory read

The read() method is used to read from a memory location, the address of the location is the offset within the memory region, rather than the absolute memory address. This allows stimulus accessing memory to be relocatable, and therefore reusable.

```
//
// memory read method prototype
//

task uvm_mem:::read(output uvm_status_e      status,          // Outcome of the write cycle
                    input  uvm_reg_addr_t   offset,         // Offset address within the memory region
                    output uvm_reg_data_t   value,          // Read data
                    input  uvm_path_e       path = UVM_DEFAULT_PATH, // Front or backdoor access
                    input  uvm_reg_map      map = null,        // Which map, memory might in be >1 map
                    input  uvm_sequence_base parent = null,    // Parent sequence
                    input  int               prior = -1,        // Priority on the target sequencer
                    input  uvm_object        extension = null, // Object allowing method extension
                    input  string             fname = "",       // Filename for messaging
                    input  int               lineno = 0);      // File line number for messaging

//
// Examples:
//

mem_ss.mem_1.read(status, 32'h1000, read_data, .parent(this)); // Using default map
//
mem_ss.mem_1.read(status, 32'h2000, read_data, .parent(this), .map(AHB_2_map)); // Using alternative map
```

## Memory write

The write() method is used to write to a memory location, and like the read method, the address of the location to be written to is an offset within the memory region.

```
//
// memory write method prototype
//

task uvm_mem:::write(output uvm_status_e      status,          // Outcome of the write cycle
                     input  uvm_reg_addr_t   offset,         // Offset address within the memory region
                     input  uvm_reg_data_t   value,          // Write data
                     input  uvm_path_e       path = UVM_DEFAULT_PATH, // Front or backdoor access
                     input  uvm_reg_map      map = null,       // Which map, memory might be in >1 map
                     input  uvm_sequence_base parent = null,   // Parent sequence
                     input  int               prior = -1,      // Priority on the target sequencer
                     input  uvm_object        extension = null, // Object allowing method extension
                     input  string            fname = "",      // Filename for messaging
                     input  int               lineno = 0);    // File line number for messaging

//
// Examples:
//

mem_ss.mem_1.write(status, 32'h1000, write_data, .parent(this)); // Using default map
//
mem_ss.mem_1.write(status, 32'h2000, write_data, .parent(this), .map(AHB_2_map)); // Using alternative map
```

## Memory burst\_read

The burst\_read() method is used to read an array of data words from a series of consecutive address locations starting from the specified offset within the memory region. The number of read accesses in the burst is determined by the size of the read data array argument passed to the burst\_read() method.

```
//
// memory burst_read method prototype
//

task uvm_mem:::burst_read(output uvm_status_e      status,          // Outcome of the write cycle
                          input  uvm_reg_addr_t   offset,         // Offset address within the memory region
                          output uvm_reg_data_t   value[],        // Read data array
                          input  uvm_path_e       path = UVM_DEFAULT_PATH, // Front or backdoor access
                          input  uvm_reg_map      map = null,       // Which map, memory might be in >1 map
                          input  uvm_sequence_base parent = null,   // Parent sequence
                          input  int               prior = -1,      // Priority on the target sequencer
                          input  uvm_object        extension = null, // Object allowing method extension
                          input  string            fname = "",      // Filename for messaging
                          input  int               lineno = 0);    // File line number for messaging

//
// Examples:
//


```

```

uvm_reg_data_t read_data[];

//
// 8 Word transfer:
//
read_data = new[8]; // Set read_data array to size 8
mem_ss.mem_1.burst_read(status, 32'h1000, read_data, .parent(this)); // Using default map
//
// 4 Word transfer from an alternative map:
//
read_data = new[4]; // Set read_data array to size 4
mem_ss.mem_1.burst_read(status, 32'h2000, read_data, .parent(this), .map(AHB_2_map));

```

## Memory burst\_write

The memory burst write() method is used to write an array of data words to a series of consecutive address locations starting from the specified offset with the memory region. The size of the data array determines the length of the burst.

```

//
// memory burst_write method prototype
//

task uvm_mem::burst_write(output uvm_status_e      status,           // Outcome of the write cycle
                          input  uvm_reg_addr_t    offset,          // Offset address within the memory region
                          input  uvm_reg_data_t    value[],        // Write data array
                          input  uvm_path_e        path = UVM_DEFAULT_PATH, // Front or backdoor access
                          input  uvm_reg_map       map = null,        // Which map, memory might be in >1 map
                          input  uvm_sequence_base parent = null,    // Parent sequence
                          input  int                prior = -1,       // Priority on the target sequencer
                          input  uvm_object         extension = null, // Object allowing method extension
                          input  string             fname = "",       // Filename for messaging
                          input  int                lineno = 0);    // File line number for messaging

//
// Examples:
//
uvm_reg_data_t write_data[];

//
// 8 Word transfer:
//
write_data = new[8]; // Set write_data array to size 8
foreach(write_data[i]) begin
  write_data[i] = i*16;
end
mem_ss.mem_1.burst_write(status, 32'h1000, write_data, .parent(this)); // Using default map
//
// 4 Word transfer from an alternative map:
//
write_data = new[4]; // Set read_data array to size 4

```

```
write_data = '{32'h55AA_55AA, 32'hAA55_AA55, 32'h55AA_55AA, 32'hAA55_AA55};  
mem_ss.mem_1.burst_write(status, 32'h2000, write_data, .parent(this), .map(AHB_2_map));
```

## Memory Burst Considerations

Exactly how the burst will be implemented is determined by the adapter class that takes care of mapping generic register transactions to/from target agent sequence\_items. If stimulus reuse with different target buses is important, care should be taken to make the burst size within a sensible range (e.g. 4, 8 or 16 beats) and to ensure that the start address is aligned to a word boundary. If there is a need to verify that a memory interface can support all potential types of burst and protocol transfer supported by a bus, then this should be done by running native sequences directly on the bus agent rather than using the memory model.

In UVM 1.0, memory burst accesses are actually broken down into single transfers by the underlying implementation of the register model adaption layer.

## Example Stimulus

The following example sequence illustrates how the memory access methods could be used to implement a simple memory test.

```
//  
// Test of memory 1  
  
// Write to 10 random locations within the memory storing the data written  
// then read back from the same locations checking against  
// the original data  
  
class mem_1_test_seq extends mem_ss_base_seq;  
  
`uvm_object_utils(mem_1_test_seq)  
  
rand uvm_reg_addr_t addr;  
  
// Buffers for the addresses and the write data  
uvm_reg_addr_t addr_array[10];  
uvm_reg_data_t data_array[10];  
  
function new(string name = "mem_1_test_seq");  
    super.new(name);  
endfunction  
  
// The base sequence sets up the register model handle  
task body;  
    super.body();  
    // Write loop  
    for(int i = 0; i < 10; i++) begin  
        // Constrain address to be within the memory range:  
        // ...  
    end  
endtask
```

```
if(!this.randomize() with {addr <= mem_ss_rm.mem_1.get_size();}) begin
  `uvm_error("body", "Randomization failed")
end

mem_ss_rm.mem_1.write(status, addr, data, .parent(this));
addr_array[i] = addr;
data_array[i] = data;
end

// Read loop
for(int i = 0; i < 10; i++) begin
  mem_ss_rm.mem_1.read(status, addr_array[i], data, .parent(this));
  if(data_array[i][31:0] != data[31:0]) begin
    `uvm_error("mem_1_test", $sformatf("Memory access error: expected %0h, actual %0h", data_array[i][31:0], data[31:0]))
  end
end

endtask: body

endclass: mem_1_test_seq
```

# Registers/SequenceExamples

To illustrate how the different register model access methods can be used from sequences to generate stimulus, this page contains a number of example sequences developed for stimulating the SPI master controller DUT.

Note that all of the example sequences that follow do not use all of the argument fields available in the methods. In particular, they do not use the map argument, since the access to the bus agent is controlled by the layering. If a register can be accessed by more than one bus interface, it will appear in several maps, possibly at different offsets. When the register access is made, the model selects which bus will be accessed. Writing the sequences this way makes it easier to reuse or retarget them in another integration scenario where they will access the hardware over a different bus infrastructure.

The examples shown are all derived from a common base sequence class template.

## Register Sequence Base Class

In order to use the register model effectively with sequences, a base sequence needs to be written that takes care of getting the handle to the model and has data and status properties which are used by the various register access methods. This base class is then extended to implement sequences which use the register model.

The data field in the base class uses a register model type called `uvm_reg_data_t` which defaults to a 64 bit variable. The status field uses a register model enumerated type called `uvm_status_e`.

```
package spi_bus_sequence_lib_pkg;

import uvm_pkg::*;
`include "uvm_macros.svh"

import spi_env_pkg::*;
import spi_reg_pkg::*;

// Base class that used by all the other sequences in the
// package:
//
// Gets the handle to the register model - spi_rm
//
// Contains the data and status fields used by most register
// access methods
//
class spi_bus_base_seq extends uvm_sequence #(uvm_sequence_item);

`uvm_object_utils(spi_bus_base_seq)

// SPI Register model:
spi_reg_block spi_rm;
// SPI env configuration object (containing a register model handle)
spi_env_config m_cfg;
```

```
// Properties used by the various register access methods:  
rand uvm_reg_data_t data; // For passing data  
uvm_status_e status; // Returning access status  
  
function new(string name = "spi_bus_base_seq");  
    super.new(name);  
endfunction  
  
// Common functionality:  
// Getting a handle to the register model  
task body;  
    if(!uvm_config_db #(spi_env_config)::get(null, get_full_name(), "spi_env_config", m_cfg)) begin  
        `uvm_error("body", "Could not find spi_env_config")  
    end  
    spi_rm = m_cfg.spi_rm;  
endtask: body  
  
endclass: spi_bus_base_seq
```

## Sequence using write

The following simple sequence illustrates the write method being used to set a register to a known value, in this case setting the SPI ss register to 0.

```
// Slave Unselect setup sequence  
//  
// Writes 0 to the slave select register  
//  
class slave_unselect_seq extends spi_bus_base_seq;  
  
`uvm_object_utils(slave_unselect_seq)  
  
function new(string name = "slave_unselect_seq");  
    super.new(name);  
endfunction  
  
task body;  
    super.body();  
    spi_rm.ss_reg.write(status, 32'h0, .parent(this));  
endtask: body  
  
endclass: slave_unselect_seq
```

## Sequence using randomize and set followed by update and get

The following sequence loads up the SPI control register with random data. In order to make the sequence reusable, the interrupt enable bit is a variable in the sequence.

The sequence randomizes the control register with a constraint that controls the size of the data word. Then the set method is used to setup the interrupt enable bit (ie) and to set the go\_bsy bit to 0. Once the desired value has been setup, the update() method is called and this causes a bus level write to occur.

The final get() call in the sequence is used to update the sequence data variable from the control registers mirror value which has been updated by the predictor at the end of the transfer. This is done so that the virtual sequence controlling the generation process can store the current configuration of the control register.

```

//  

// Ctrl set sequence - loads one control params  

//                      but does not set the go bit  

//  

class ctrl_set_seq extends spi_bus_base_seq;  

`uvm_object_utils(ctrl_set_seq)  

function new(string name = "ctrl_set_seq");  

    super.new(name);  

endfunction  

// Controls whether interrupts are enabled or not  

bit int_enable = 0;  

task body;  

    super.body;  

    // Constrain to interesting data length values  

    if(!spi_rm.ctrl_reg.randomize() with {char_len.value inside {0, 1, [31:33], [63:65], [95:97], 126, 127};}) begin  

        `uvm_error("body", "Control register randomization failed")  

    end  

    // Set up interrupt enable  

    spi_rm.ctrl_reg.ie.set(int_enable);  

    // Don't set the go_bsy bit  

    spi_rm.ctrl_reg.go_bsy.set(0);  

    // Write the new value to the control register  

    spi_rm.ctrl_reg.update(status, .path(UVM_FRONTDOOR), .parent(this));  

    // Get a copy of the register value for the SPI agent  

    data = spi_rm.ctrl_reg.get();  

endtask: body  

endclass: ctrl_set_seq

```

## Sequence using read and write, peek and poke

The SPI register test sequence illustrates the use of the read() and write() methods, followed by the peek() and poke() methods. It also illustrates a few other features of the register model which can prove useful for similar types of tests.

The register test sequence starts by doing a read from all the registers to check their reset values, then it writes a random value to each register in a random order, followed by a read that checks that the value read back is the same as the value written. In order to access all the registers in the SPI block, the block level get\_registers() method is called. This returns an array of handles for all the registers in the block, and subsequent accesses to the registers are made using this array.

In the reset test, the reset value for each register is copied into the ref\_data variable using the get\_reset() method. Then the register is read and the returned data value compared against the ref\_data variable.

In the write/read test, the base sequences data variable is randomized and then written to the selected register. Then the register handle array is shuffled and the get() method is used to copy the registers mirrored value into ref\_data to be compared against the real hardware value via the read() method.

In the peek and poke test, the same test strategy is used, but this time using the backdoor access.

Note that in all this activity, the register address is not used. Only in the write loop is there a check on the name of the register to ensure that if it is the control register that the bit that initiates the SPI transfer is not set.

```

//  

// This is a register check sequence  

//  

// It checks the reset values  

//  

// Then it writes random data to each of the registers  

// and reads back to check that it matches  

//  

class check_regs_seq extends spi_bus_base_seq;  

`uvm_object_utils(check_regs_seq)  

function new(string name = "check_regs_seq");  

  super.new(name);  

endfunction  

uvm_reg spi_regs[$];  

uvm_reg_data_t ref_data;  

task body;  

  super.body;  

  spi_rm.get_registers(spi_regs);  

  // Read back reset values in random order  

  spi_regs.shuffle();  

  foreach(spi_regs[i]) begin  

    ref_data = spi_regs[i].get_reset();  

  end
endtask

```

```
spi_regs[i].read(status, data, .parent(this));

if(ref_data != data) begin
  `uvm_error("REG_TEST_SEQ:", $sformatf("Reset read error for %s: Expected: %0h Actual: %0h", spi_regs[i].get_name(), ref_data, data))
end
end

// Write random data and check read back (10 times)
repeat(10) begin

  spi_regs.shuffle();
  foreach(spi_regs[i]) begin
    assert(this.randomize());
    if(spi_regs[i].get_name() == "ctrl") begin
      data[8] = 0;
    end
    spi_regs[i].write(status, data, .parent(this));
  end
  spi_regs.shuffle();
  foreach(spi_regs[i]) begin
    ref_data = spi_regs[i].get();
    spi_regs[i].read(status, data, .parent(this));
    if(ref_data != data) begin
      `uvm_error("REG_TEST_SEQ:", $sformatf("get/read: Read error for %s: Expected: %0h Actual: %0h", spi_regs[i].get_name(), ref_data, data))
    end
  end
end

// Repeat with back door accesses
repeat(10) begin
  spi_regs.shuffle();
  foreach(spi_regs[i]) begin
    assert(this.randomize());
    if(spi_regs[i].get_name() == "ctrl") begin
      data[8] = 0;
    end
    spi_regs[i].poke(status, data, .parent(this));
  end
  spi_regs.shuffle();
  foreach(spi_regs[i]) begin
    ref_data = spi_regs[i].get();
    spi_regs[i].peek(status, data, .parent(this));
    if(ref_data[31:0] != data[31:0]) begin
      `uvm_error("REG_TEST_SEQ:", $sformatf("poke/peek: Read error for %s: Expected: %0h Actual: %0h", spi_regs[i].get_name(), ref_data, data))
    end
  end
end
```

```

    end

    spi_regs[i].read(status, data, .parent(this));

end

end

endtask: body

endclass: check_regs_seq

```

## Sequence using mirror

The mirror() method causes a read access to take place which updates the mirrored value in the register, it does not return the read data. This can be useful to either re-sync the register model with the hardware state. The mirror() method can be used, together with a predict() call, to check that the data read back from a register matches the expected data.

The following sequence uses the mirror() method to check that the data read back from the target hardware registers matches the expected values which have been set in the register mirrored values. In this case, the mirrored values have been set via the testbench scoreboard according to the data monitored on the input bus. The mirror() method call has its check field set to UVM\_CHECK which ensures that the data actually read back matches that predicted by the scoreboard. If the check field is at its default value, the mirror() method would simply initiate a bus read cycle which would result in the external predictor updating the mirror value.

```

// Data unload sequence - reads back the data rx registers
// all of them in a random order
//

class data_unload_seq extends spi_bus_base_seq;

`uvm_object_utils(data_unload_seq)

uvm_reg data_regs[];

function new(string name = "data_unload_seq");
    super.new(name);
endfunction

task body;
    super.body();
    // Set up the data register handle array
    data_regs = '{spi_rm.rxtx0_reg, spi_rm.rxtx1_reg, spi_rm.rxtx2_reg, spi_rm.rxtx3_reg};
    // Randomize access order
    data_regs.shuffle();
    // Use mirror in order to check that the value read back is as expected
    foreach(data_regs[i]) begin
        data_regs[i].mirror(status, UVM_CHECK, .parent(this));
    end
endtask

```

```
  end
endtask: body

endclass: data_unload_seq
```

Note that this example of a sequence interacting with a scoreboard is not a recommended approach, it is provided as a means of illustrating the use of the mirror() method.

# Registers/BuiltInSequences

The UVM package contains a library of automatic test sequences which are based on the register model. These sequences can be used to do basic tests on registers and memory regions within a DUT. The automatic tests are aimed at testing basic functionality such as checking register reset values are correct or that the read-write data paths are working correctly. One important application of these sequences is for quick sanity checks when bringing up a sub-system or SoC design where a new interconnect or address mapping scheme needs to be verified.

Registers and memories can be opted out of these auto tests by setting an individual "DO\_NOT\_TEST" attribute which is checked by the automatic sequence as runs. An example of where such an attribute would be used is a clock control register where writing random bits to it will actually stop the clock and cause all further DUT operations to fail.

The register sequences which are available within the UVM package are summarised in the following tables.

**Note** that any of the automatic tests can be disabled for a given register or memory by the **NO\_REG\_TEST** attribute, or for memories by the **NO\_MEM\_TEST** attribute. The disable attributes given in the table are specific to the sequences concerned.

## Register Built In Sequences

Sequence	Disable Attribute	Block Level	Register Level	Description
uvm_reg_hw_reset_seq	NO_REG_HW_RESET_TEST	Yes	Yes	Checks that the Hardware register reset value matches the value specified in the register model
uvm_reg_single_bit_bash_seq	NO_REG_BIT_BASH_TEST	No	Yes	Writes, then check-reads 1's and 0's to all bits of the selected register that have read-write access
uvm_reg_bit_bash_seq	NO_REG_BIT_BASH_TEST	Yes	No	Executes the uvm_reg_single_bit_bash_seq for each register in the selected block and any sub-blocks it may contain
uvm_reg_single_access_seq	NO_REG_ACCESS_TEST	No	Yes	Writes to the selected register via the frontdoor, checks the value is correctly written via the backdoor, then writes a value via the backdoor and checks that it can be read back correctly via the frontdoor. Repeated for each address map that the register is present in. Requires that the backdoor hdl_path has been specified
uvm_reg_access_seq	NO_REG_ACCESS_TEST	Yes	No	Executes the uvm_reg_single_access_seq for each register accessible via the selected block
uvm_reg_shared_access_seq	NO_SHARED_ACCESS_TEST	No	Yes	For each map containing the register, writes to the selected register in one map, then check-reads it back from all maps from which it is accessible. Requires that the selected register has been added to multiple address maps.

Some of the register test sequences are designed to run on an individual register, whereas some are block level sequences which go through each accessible register and execute the single register sequence on it.

## Memory Built In Sequences

Sequence	Disable Attributes	Block Level	Memory Level	Description
uvm_mem_single_walk_seq	NO_MEM_WALK_TEST	No	Yes	Writes a walking pattern into each location in the range of the specified memory, then checks that is read back with the expected value
uvm_mem_walk_seq	NO_MEM_WALK_TEST	Yes	No	Executes uvm_mem_single_walk_seq on all memories accessible from the specified block
uvm_mem_single_access_seq	NO_MEM_ACCESS_TEST	No	Yes	For each location in the range of the specified memory: Writes via the frontdoor, checks the value written via the backdoor, then writes via the backdoor and reads back via the front door. Repeats test for each address map containing the memory. Requires that the backdoor hdl_path has been specified.
uvm_mem_access_seq	NO_MEM_ACCESS_TEST	Yes	No	Executes uvm_mem_single_access_seq for each memory accessible from the specified block
uvm_mem_shared_access_seq	NO_SHARED_ACCESS_TEST	No	Yes	For each map containing the memory, writes to all memory locations and reads back from all using each of the address maps. Requires that the memory has been added to multiple address maps.

Like the register test sequences, the tests either run on an individual memory basis, or on all memories contained within a block. Note that the time taken to execute a memory test sequence could be lengthy with a large memory range.

## Aggregated Register And Memory Built In Sequences

These sequences run at the block level only

Sequence	Disable Attributes	Description
uvm_reg_mem_shared_access_seq	NO_SHARED_ACCESS_TEST	Executes the uvm_reg_shared_access_seq on all registers accessible from the specified block. Executes the uvm_mem_shared_access_seq on all memories accessible from the specified block
uvm_reg_mem_built_in_seq	All of the above	Executes all of the block level auto-test sequences
uvm_reg_mem_hdl_paths_seq	None	Used to test that any specified HDL paths defined within a block are accessible by the backdoor access. The check is only performed on registers or memories which have HDL paths declared.

## Setting An Attribute

In order to set an auto-test disable attribute on a register, you will need to use the UVM resource\_db to set a bit with the attribute string, giving the path to the register or the memory as the scope variable. Since the UVM resource database is used, the attributes can be set from anywhere in the testbench at any time. However, the recommended approach is to set the attributes as part of the register model, this will most likely be done by specifying the attribute via the register model generators specification input file.

The following code excerpt shows how attributes would be implemented in a register model.

```

// From the build() method of the memory sub-system (mem_ss) block:
function void build();
  //
  // ....
  //
  // Example use of "dont_test" attributes:
  // Stops mem_1_offset reset test
  uvm_resource_db #(bit)::set({"REG::", this.mem_1_offset.get_full_name()}, "NO_REG_HW_RESET_TEST", 1);
  // Stops mem_1_offset bit-bash test
  uvm_resource_db #(bit)::set({"REG::", this.mem_1_offset.get_full_name()}, "NO_REG_BIT_BASH_TEST", 1);
  // Stops mem_1 being tested with the walking auto test
  uvm_resource_db #(bit)::set({"REG::", this.mem_1.get_full_name()}, "NO_MEM_WALK_TEST", 1);
  lock_model();
endfunction: build

```

This shows how the same could be achieved from a sequence:

```

//
// From within a sequence where the mem_ss_rm handle is set in the base_class:
//
task body;
  super.body();
  // Disable mem_2 walking auto test
  uvm_resource_db #(bit)::set({"REG::", mem_ss_rm.mem_1.get_full_name()}, "NO_MEM_WALK_TEST", 1);
  //
  // ...
  //
endtask: body

```

Note that once an attribute has been set in the UVM resource database, it cannot be 'unset'. This means that successive uses of different levels of disabling within sequences may produce unwanted accumulative effects.

## Built In Sequence Example

The example sequence is from a testbench for a memory sub-system which contains some registers which set up the address decoding for the sub-system memory arrays and then control accesses to them. It uses 3 of the built-in test sequences, two to test the register and the other to test all of the memory arrays in the sub-system.

In order for any of the UVM automatic tests to run, they need to have their register model handle assigned to the register model being used. They all use the name 'model' for the register model handle. This is illustrated in the example.

```

//
// Auto test of the memory sub-system using the built-in
// automatic sequences:
//
class auto_tests extends mem_ss_base_seq;

```

```
`uvm_object_utils(auto_tests)

function new(string name = "auto_tests");
    super.new(name);
endfunction

task body;
    // Register reset test sequence
    uvm_reg_hw_reset_seq rst_seq = uvm_reg_hw_reset_seq::type_id::create("rst_seq");
    // Register bit bash test sequence
    uvm_reg_bit_bash_seq reg_bash = uvm_reg_bit_bash_seq::type_id::create("reg_bash");
    // Initialise the memory mapping registers in the sub-system
    mem_setup_seq setup = mem_setup_seq::type_id::create("setup");
    // Memory walk test sequence
    uvm_mem_walk_seq walk = uvm_mem_walk_seq::type_id::create("walk");

    super.body(); // Gets the register model handle
    // Set the register model handle in the built-in sequences
    rst_seq.model = mem_ss_rm;
    walk.model = mem_ss_rm;
    reg_bash.model = mem_ss_rm;

    // Start the test sequences:
    //
    // Register reset:
    rst_seq.start(m_sequencer);
    // Register bit-bash
    reg_bash.start(m_sequencer);
    // Set up the memory sub-system
    setup.start(m_sequencer);
    // Memory walk test
    walk.start(m_sequencer);

endtask: body

endclass: auto_tests
```

## Example Download

The code for this example can be downloaded via the following link:

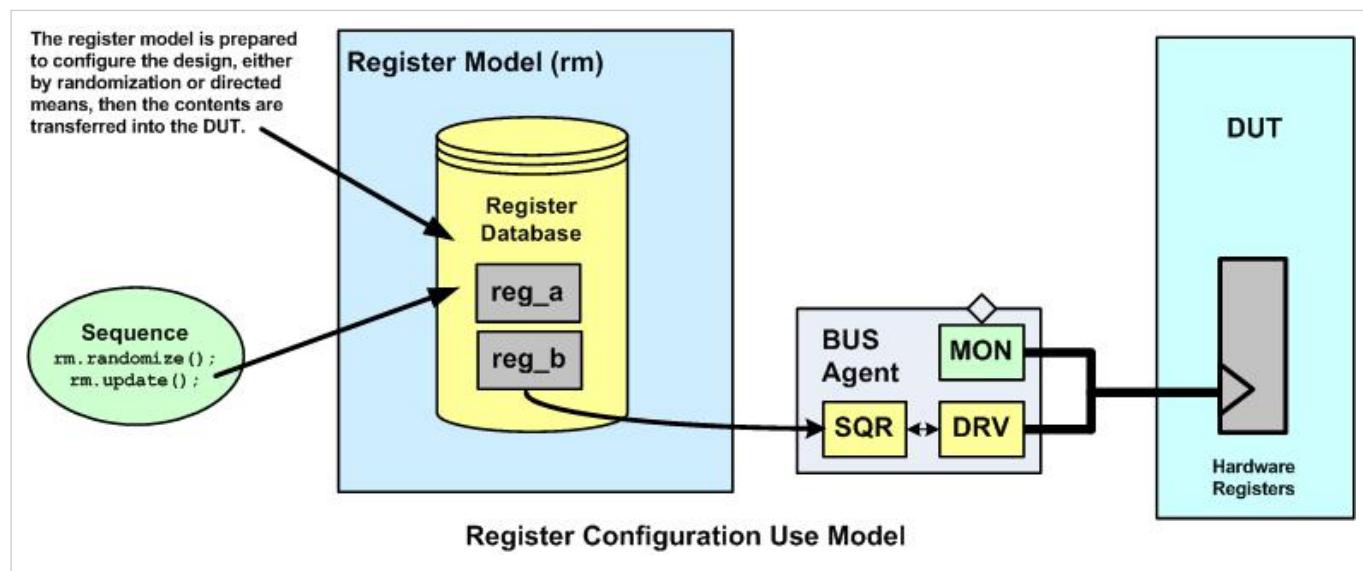
( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Registers/Configuration

During verification a programmable hardware device needs to be configured to operate in different modes. The register model can be used to automate or to semi-automate this process.

The register model contains a shadow of the register state space for the DUT which is kept up to date as the design is configured using bus read and write cycles. One way to configure the design in a testbench is to apply reset to the design and then go through a programming sequence which initialises the design for a particular mode of operation.

In real life, a programmable device might be used for a while in one mode of operation, then reconfigured to be used in another mode and the non re-programmed registers will effectively have random values. Always initialising from reset has the short coming that the design always starts from a known state and it is possible that a combination of register values that causes a design failure would be missed. However, if the register map is randomized at the beginning of the simulation and the randomized contents of the register map are written to the DUT before configuring it into the desired mode, it is possible to emulate the conditions that would exist in a 'mid-flight' re-configuration.



The register model can be used to configure the design by creating a configuration state 'off-line' using whatever mixture of constrained randomization or directed programming is convenient. If the register model desired values are updated, then the transfer of the configuration to the DUT can be done using the update() method, this will transfer any new values that need to be written to the hardware in the order they are declared in the register model.

The transfer of the register model to the DUT can be done either in an ordered way, or it can be done in a random order. Some designs require that at least some of the register programming is done in a particular order.

In order to transfer the data in a random order, the registers in the model should be collected into an array and then the array should be shuffled:

```
//
// Totally random configuration
//
task body;
  uvm_reg spi_regs[];
```

```

super.body();
spi_rm.get_registers(spi_regs);
if(!spi_rm.randomize()) begin
  `uvm_error("body", "spi_rm randomization failed")
end
spi_regs.shuffle(); // Randomly re-order the array
foreach(spi_regs[i]) begin
  spi_regs[i].update(); // Only change the reg if required
end
endtask: body

```

Here is an example of a sequence that configures the SPI using the register model. Note that it uses constraints to configure the device within a particular range of operation, and that the write to the control register is a setup write which will be followed by an enabling write in another sequence.

```

// Sequence to configure the SPI randomly
//
class SPI_config_seq extends spi_bus_base_seq;

`uvm_object_utils(SPI_config_seq)

function new(string name = "SPI_config_seq");
  super.new(name);
endfunction

bit interrupt_enable;

task body;
  super.body();

  // Randomize the register model to get a new config
  // Constraining the generated value within ranges
  if(!spi_rm.randomize() with {spi_rm.ctrl_reg.go_bsy.value == 0;
    spi_rm.ctrl_reg.ie.value == interrupt_enable;
    spi_rm.ss_reg.cs.value != 0;
    spi_rm.ctrl_reg.char_len.value inside {0, 1, [31:33], [63:65], [95:97], 126, 127};
    spi_rm.divider_reg.ratio.value inside {16'h0, 16'h1, 16'h2, 16'h4, 16'h8, 16'h10, 16'h20, 16'h40, 16'h80};
  }) begin
    `uvm_error("body", "spi_rm randomization failure")
  end
  // This will write the generated values to the HW registers
  spi_rm.update(status, .path(UVM_FRONTDOOR), .parent(this));
  data = spi_rm.ctrl_reg.get();
endtask: body

```

```
endclass: SPI_config_seq
```

A DUT could be reconfigured multiple times during a test case in order to find unintended interactions between register values.

# Registers/Scoreboarding

---

The UVM register model shadows the current configuration of a programmable DUT and this makes it a valuable resource for scoreboards that need to be aware of the current DUT state. The scoreboard references the register model to either adapt its configuration to match the current state of a programmable DUT or to adapt its algorithms in order to make the right comparisons. For instance, checking that a communications device with a programmable packet configuration has transmitted or received the right data requires the scoreboard to know the current data format.

The UVM register model will contain information that is useful to the scoreboard:

- DUT Configuration, based on the register state
- Data values, based on the current state of buffer registers

The UVM register model can also be used by the scoreboard to check DUT behaviour:

- A register model value can be set up to check for the correct expected value
- A backdoor access can check the actual state of the DUT hardware - This can be useful if there is volatile data that causes side effects if it is read via the front door.

## Register Model Access

In order to reference the contents of the register model, the scoreboard will require its handle. This can be assigned either from a configuration object or by using a resource from the UVM resource database. Once this handle is assigned, then the contents of the register data base can be accessed. Field or register values can either be accessed by reference using a path to the mirrored field value (e.g. `spi_rm.ctrl.ie.value`) or by calling the `get_mirrored_value()` access method (e.g. `spi_rm.ctrl.ie.get_mirrored_value()`). This configuration information can then be used to inform decisions in the scoreboard logic.

## Checking DUT Behaviour

DUT register contents can be checked in one of several ways.

The simplest way is to compare an observed output against the original data. For instance, a scoreboard for a parallel to serial converter function could compare the data in an observed serial output packet against the data in the transmit buffer registers.

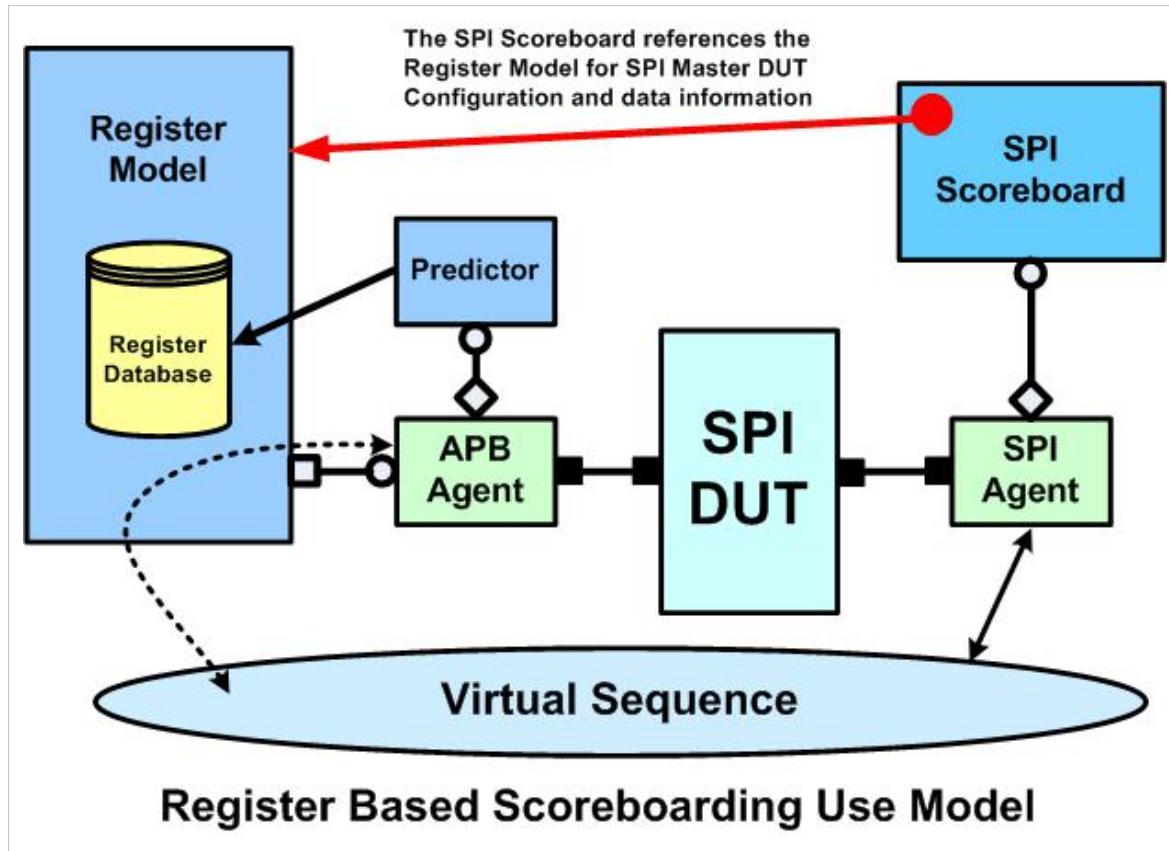
Another way is to use the register model to check the contents of the DUT against their expected contents. For instance, in the case of a serial to parallel converter, the scoreboard could determine the expected contents of the receive buffer, and use a back door `peek()` access of the hardware receive buffer to determine that the correct data has been received.

An alternative to using `peek()` would be to use the `predict()` register access method to set the mirrored values and then access the DUT registers using the `mirror()` access function with its `check` argument set. The `mirror()` call will detect and report any errors due to a mismatch between the expected and actual data. The `mirror()` accesses could be made from the scoreboard, or they could be executed from a supporting sequence.

**Note:** For information on `peek()`, `mirror()` and other register model access methods mentioned on this page, see the material on register accesses.

## Example register model based scoreboard

The SPI scoreboard uses the SPI register model in several of the ways described above. In order to scoreboard the SPI master DUT, it is necessary to check its configuration. In this case, the SPI master DUT is capable of transferring a variable number of bits in and out of the data buffer in either LSB or MSB first order. It can also transmit on different clock edges and sample receive data on different clock edges. This configuration information is looked up in the SPI register model so that the scoreboard logic is configured correctly.



### Functional Overview

The scoreboard reacts to analysis transactions from the SPI agent. These correspond to a completed transfer, and the transmitted (Master Out Slave In -MOSI) data is compared against the data in the transmit buffer in the register model before the received (Master In Slave Out - MISO) data is compared against the data peeked from the DUT receive buffer. The RXTX register mirrored value is also updated using the predict() method, this allows any subsequent front door access using a checking mirror() method call to check the read data path from these registers.

The SPI scoreboard also checks that the SPI chip select values used during the transfer match the value programmed in the slave select (SS) register.

The following excerpts from the scoreboard code illustrate how the scoreboard references the register model.

## Register Model Handle

The SPI scoreboard needs to refer to the spi\_rm register model and contains a handle to do so. This handle is assigned in its containing env from the register model handle in the envs configuration object:

```
function void spi_env::connect();
//
  if(m_cfg.has_spi_scoreboard) begin
    m_spi_agent.ap.connect(m_scoreboard.spi.analysis_export); // SPI Monitor
    m_scoreboard.spi_rm = m_cfg.spi_rm; // SPI Register Model
  end
//
endfunction: connect
```

## Referring to the Register Model To Check SPI TX (MOSI) Data

The SPI master DUT can transmit up to 128 bits of data, and the transmit order can be either LSB or MSB first. The data can be changed on either the rising or the falling edge of the SPI clock. The SPI agent monitors the SPI data transfer and sends an analysis transaction every time a SPI character transfer completes. The scoreboard uses the content of the SPI analysis transaction, together with the configuration details from the register model to compare the observed transmit data against the data written into the TX data registers.

```
forever begin
  error = 0;
  spi.get(item);
  no_transfers++;
  // Get the character length from the register model
  //
  // char_len is the character length field in the ctrl register
  //
  bit_cnt = spi_rm.ctrl_reg.char_len.get_mirrored_value();
  // Corner case for bit count equal to zero:
  if(bit_cnt == 8'b0) begin
    bit_cnt = 128;
  end
  // Get the transmit data from the register model
  //
  // The 128 bits of tx data is in 4 registers rxtx0-rxtx3
  //
  tx_data[31:0] = spi_rm.rxtx0_reg.get_mirrored_value();
  tx_data[63:32] = spi_rm.rxtx1_reg.get_mirrored_value();
  tx_data[95:64] = spi_rm.rxtx2_reg.get_mirrored_value();
  tx_data[127:96] = spi_rm.rxtx3_reg.get_mirrored_value();

  // Find out if the tx (mosi) data is sampled on the neg or pos edge
  //
  // The ctrl register tx_neg field contains this info
```

```
//  
// The SPI analysis transaction (item) contains samples for both edges  
//  
if(spi_rm.ctrl_reg.tx_neg.get_mirrored_value() == 1) begin  
    mosi_data = item.nedge_mosi; // To be compared against write data  
end  
else begin  
    mosi_data = item.pedge_mosi;  
end  
//  
// Compare the observed MOSI bits against the tx data written to the SPI DUT  
//  
// Find out whether the MOSI data is transmitted LSB or MSB first - this  
// affects the comparison  
if(spi_rm.ctrl_reg.lsb.get_mirrored_value() == 1) begin // LSB first  
    for(int i = 0; i < bit_cnt; i++) begin  
        if(tx_data[i] != mosi_data[i]) begin  
            error = 1;  
        end  
    end  
    if(error == 1) begin  
        `uvm_error("SPI_SB_MOSI_LSB:", $sformatf("Expected mosi value %0h actual %0h", tx_data, mosi_data))  
    end  
end  
else begin // MSB first  
    for(int i = 0; i < bit_cnt; i++) begin  
        if(tx_data[i] != mosi_data[(bit_cnt-1) - i]) begin  
            error = 1;  
        end  
    end  
    if(error == 1) begin // Need to reverse the mosi_data bits for the error message  
        rev_miso = 0;  
        for(int i = 0; i < bit_cnt; i++) begin  
            rev_miso[(bit_cnt-1) - i] = mosi_data[i];  
        end  
        `uvm_error("SPI_SB_MOSI_MSB:", $sformatf("Expected mosi value %0h actual %0h", tx_data, rev_miso))  
    end  
end  
if(error == 1) begin  
    no_tx_errors++;  
end  
//  
// TX Data checked
```

### Referring to the Register Model To Check SPI RX (MISO) Data

The SPI slave (RX) data will follow the same format as the transmit data, but it could be sampled on the opposite edge of the SPI clock, this is determined by a control bit field in the SPI control register. The scoreboard looks up this information, manipulates the MISO data in the SPI analysis transaction as necessary to get it in the right order and then compares it against the value read back by a series of back door peek()s from the SPI data registers.

Although not strictly necessary, the scoreboard calls the predict() method for each of the data registers using the data observed via the peek() method. This sets things up so that if a sequence makes a mirror() call to these registers with a check enabled, the return data path will be checked.

```
// RX Data check

    // Reset the error bit
    error = 0;

    // Check the miso data (RX)
    //

    // Look up in the register model which edge the RX data should be sampled on
    //

    if(spi_rm.ctrl_reg.rx_neg.get_mirrored_value() == 1) begin
        miso_data = item.pedge_miso;
    end
    else begin
        miso_data = item.nedge_miso;
    end

    // Reverse the order of the observed RX data bits if MSB first
    //

    // Determine this by looking up the ctrl.lsb in the register model
    //

    if(spi_rm.ctrl_reg.lsb.get_mirrored_value() == 0) begin // MSB
        // reverse the bits lsb -> msb, and so on
        rev_miso = 0;
        for(int i = 0; i < bit_cnt; i++) begin
            rev_miso[(bit_cnt-1) - i] = miso_data[i];
        end
        miso_data = rev_miso;
    end

    // The following sets up the rx data so that it is
    // bit masked according to the no of bits
    rx_data = spi_rm.rxtx0_reg.get_mirrored_value();
    // Peek the RX data in the hardware and compare against the observed RX data
    spi_rm.rxtx0_reg.peek(status, spi_peek_data);
    for(int i = 0; ((i < 32) && (i < bit_cnt)); i++) begin
        rx_data[i] = miso_data[i];
        if(spi_peek_data[i] != miso_data[i]) begin
            error = 1;
        end
    end
}
```

```

`uvm_error("SPI_SB_RXD:", $sformatf("Bit%0d Expected RX data value %0h actual %0h", i, spi_peek_data[31:0], miso_data))

end

end

// Get the register model to check that the data it next reads back from this
// register is as predicted - this is done by a sequence calling the mirror()
// method with a check enabled

//

// This is somewhat redundant given the earlier peek check, but it does check the
// read back path

assert(spi_rm.rxtx0_reg.predict(rx_data));

// Repeat for any remaining bits with the rxtx1, rxtx2, rxtx3 registers

```

## Checking the SPI Chip Selects

The final check in the scoreboard is that the observed value of the SPI slave chip select lines corresponds to the value programmed in the SPI SS registers cs (chip select) field.

```

// Check the chip select lines
//
// Compare the programmed value of the SS register (i.e. its cs field) against
// the cs bit pattern observed on the SPI bus
//
if(spi_rm.ss_reg.cs.get_mirrored_value() != {56'h0, ~item.cs}) begin
  `uvm_error("SPI_SB_CS:", $sformatf("Expected cs value %b actual %b", spi_rm.ss_reg.cs.get(), ~item.cs))
  no_cs_errors++;
end

```

The scoreboard code can be found in the *spi\_scoreboard.svh* file in the env sub-directory of the SPI block level testbench example which can be downloaded via the link below:

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# Registers/FunctionalCoverage

## Register Based Functional Coverage Overview

The UVM supports the collection of functional coverage based on register state in three ways:

- Automatic collection of register coverage based on covergroups inside the register model on each access
- Controlled collection of register coverage based on covergroups inside the register model by calling a method from outside the register model
- By reference, from an external covergroup that samples register value via a register model handle

Most register model generators allow users to specify the automatic generation of cover groups based on bit field or register content. These are fine if you have a narrow bit field and you are interested in all the states that the field could take, but they quickly loose value and simply add simulation overhead for minimal return. In order to gather register based functional coverage that is meaningful, you will need to specify coverage in terms of a cross of the contents of several registers and possibly non register signals and/or variables. Your register model generation may help support this level of complexity but, if not, it is quite straight-forward to implement an external functional coverage collection component that references the register model.

The recommended approach is to use an external covergroup that samples register values via the register model handle.

## Controlling Register Model Functional Coverage Collection

A register model may contain many covergroups and this has the potential to have a serious impact on simulation performance. Therefore there are various inter-locks built into the register model which allow you to specify which type of coverage model you want to use and to enable or disable coverage collection during the course of a test case. A bit mapped enumerated type is used to enable the different coverage models and the available options are:

enum value	Coverage Enabled
UVM_NO_COVERAGE	None, all coverage disabled
UVM_CVR_REG_BITS	Collect coverage for bits read from or written to in registers
UVM_CVR_ADDR_MAP	Collect coverage for addresses read from or written to in address maps
UVM_CVR_FIELD_VALS	Collect coverage for the values stored in register fields
UVM_CVR_ALL	Collect all coverage

The bit mapped enumeration allows several coverage models to be enabled in one assignment by logically ORing several different values - e.g. `set_coverage(UVM_CVR_ADDR_MAP + UVM_CVR_FIELD_VALS)`

A register model can contain coverage groups which have been assigned to each of the active categories and the overall coverage for the register model is set by a static method in the `uvm_reg` class called `include_coverage()`. This method should be called before the register model is built, since it creates an entry in the resource database which the register model looks up during the execution of its `build()` method to determine which covergroups to build.

```

//  

//From the SPI test base  

//  

// Build the env, create the env configuration  

// including any sub configurations and assigning virtual interfaces

```

```
function void spi_test_base::build_phase(uvm_phase build);
  // env configuration
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // Register model
  // Enable all types of coverage available in the register model
  uvm_reg::include_coverage("*", UVM_CVR_ALL);
  // Create the register model:
  spi_rm = spi_reg_block::type_id::create("spi_rm");
  // Build and configure the register model
  spi_rm.build();
```

As the register model is built, coverage sampling is enabled for the different coverage categories that have been enabled. The coverage sampling for a category of covergroups within a register model hierarchical object can then be controlled using the `set_coverage()` method in conjunction with the `has_coverage()` method (which returns a value corresponding to the coverage categories built in the scope) and the `get_coverage()` method (which returns a value corresponding to the coverage model types that are currently being actively sampled).

For more detail on how to implement a register model so that it complies with the build and control structure for covergroups see [ModelCoverage](#).

## Register Model Coverage Sampling

The covergroups within the register model will in most cases be defined by the model specification and generation process and the end user may not know how they are implemented. The covergroups within the register model can be sampled in one of two ways.

Some of the covergroups in the register model are sampled as a side-effect of a register access and are therefore automatically sampled. For each register access, the automatic coverage sampling occurs in the register and in the block that contains it. This type of coverage is important for getting coverage data on register access statistics and information which can be related to access of a specific register.

Other covergroups in the register model are only sampled when the testbench calls the `sample_values()` method from a component or a sequence elsewhere in the testbench. This allows more specialised coverage to be collected. Potential applications include:

- Sampling register state (DUT configuration) when a specific event such as an interrupt occurs
- Sampling register state only when a particular register is written to

## Referencing The Register Model Data In External Functional Coverage Monitors (Recommended)

An alternative way of implementing register based functional coverage is to build a functional coverage monitor component separate from the register model, but to sample values within the register model. The advantages of this approach are:

- The covergroup(s) within the external monitor can be developed separately from the register model implementation
- The sampling of the covergroup(s) can be controlled more easily
- It is possible to mix, or cross, sampled contents of the register model with the sampled values of other testbench variables

The following example shows a functional coverage monitor from the SPI testbench which references the SPI register model.

```
class spi_reg_functional_coverage extends uvm_subscriber #(apb_seq_item);

`uvm_component_utils(spi_reg_functional_coverage)

logic [4:0] address;
bit wnr;

spi_reg_block spi_rm;

// Checks that the SPI master registers have
// all been accessed for both reads and writes
covergroup reg_rw_cov;
  option.per_instance = 1;
  ADDR: coverpoint address {
    bins DATA0 = {0};
    bins DATA1 = {4};
    bins DATA2 = {8};
    bins DATA3 = {5'hC};
    bins CTRL  = {5'h10};
    bins DIVIDER = {5'h14};
    bins SS = {5'h18};
  }
  CMD: coverpoint wnr {
    bins RD = {0};
    bins WR = {1};
  }
  RW_CROSS: cross CMD, ADDR;
endgroup: reg_rw_cov

// 
// Checks that we have tested all possible modes of operation
// for the SPI master
```

```
//  
// Note that the field value is 64 bits wide, so only the relevant  
// bit(s) are used  
covergroup combination_cov;  
  
option.per_instance = 1;  
  
ACS: coverpoint spi_rm.ctrl_reg.acs.value[0];  
IE: coverpoint spi_rm.ctrl_reg.ie.value[0];  
LSB: coverpoint spi_rm.ctrl_reg.lsb.value[0];  
TX_NEG: coverpoint spi_rm.ctrl_reg.tx_neg.value[0];  
RX_NEG: coverpoint spi_rm.ctrl_reg.rx_neg.value[0];  
// Suspect character lengths - there may be more  
CHAR_LEN: coverpoint spi_rm.ctrl_reg.char_len.value[6:0] {  
    bins LENGTH[] = {0, 1, [31:33], [63:65], [95:97], 126, 127};  
}  
CLK_DIV: coverpoint spi_rm.divider_reg.ratio.value[7:0] {  
    bins RATIO[] = {16'h0, 16'h1, 16'h2, 16'h4, 16'h8, 16'h10, 16'h20, 16'h40, 16'h80};  
}  
COMB_CROSS: cross ACS, IE, LSB, TX_NEG, RX_NEG, CHAR_LEN, CLK_DIV;  
endgroup: combination_cov  
  
extern function new(string name = "spi_reg_functional_coverage", uvm_component parent = null);  
extern function void write(T t);  
  
endclass: spi_reg_functional_coverage  
  
function spi_reg_functional_coverage::new(string name = "spi_reg_functional_coverage", uvm_component parent = null);  
    super.new(name, parent);  
    reg_rw_cov = new();  
    combination_cov = new();  
endfunction  
  
function void spi_reg_functional_coverage::write(T t);  
    // Register coverage first  
    address = t.addr[4:0];  
    wnr = t.we;  
    reg_rw_cov.sample();  
    // Sample the combination covergroup when go_bsy is true  
    if(address == 5'h10)  
        begin  
            if(wnr) begin  
                if(t.data[8] == 1) begin  
                    combination_cov.sample(); // TX started
```

```
    end
  end
endfunction: write
```

## Coding Guideline: Wrap covergroups within uvm\_objects

A covergroup should be implemented within a wrapper class derived from an uvm\_object.

### Justification:

Wrapping a covergroup in this way has the following advantages:

- The uvm\_object can be constructed at any time - and so the covergroup can be brought into existence at any time, this aids conditional deferred construction\*
- The covergroup wrapper class can be overridden from the factory, which allows an alternative coverage model to be substituted if required
- This advantage may become more relevant when different active phases are used in future.

### Example:

```
class covergroup_wrapper extends uvm_object;
`uvm_object_utils(covergroup_wrapper)

covergroup cg (string name) with function sample(my_reg reg, bit is_read);
  option.name = name;
  CHAR_LEN: coverpoint reg.char_len {
    bins len_5 = {2'b00};
    bins len_6 = {2'b01};
    bins len_7 = {2'b10};
    bins len_8 = {2'b11};
  }
  PARITY: coverpoint reg.parity {
    bins parity_on = {1'b1};
    bins parity_off = {1'b0};
  }
  ALL_OPTIONS: cross CHAR_LEN, PARITY;
endgroup: cg

function new(string name = "covergroup_wrapper");
  super.new(name);
  cg = new();
endfunction

function void sample(my_reg reg_in, bit is_read_in);
  cg.sample(my_reg reg_in, bit is_read_in);
endfunction
```

```
endfunction: sample

endclass: covergroup_wrapper
```

For another example of this technique see the material on CoverageModelSwap

---

# Testbench Acceleration through Co-Emulation

---

## Emulation

Learn all about methodology related to Veloce/TBX Emulation on UVM

### Emulation Chapter contents:

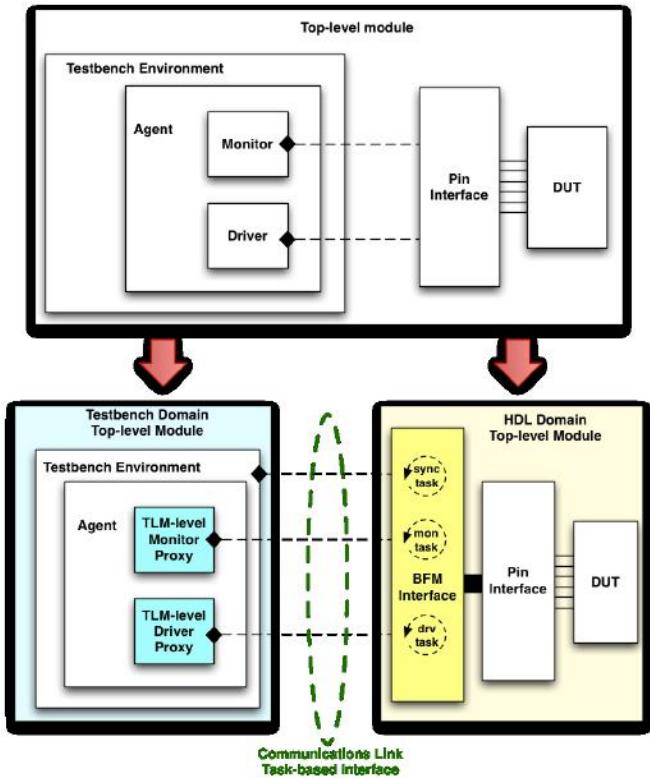
- [Emulation \(this page\)](#)
- [Emulation/SeparateTopLevels](#)
- [Emulation/SplitTransactors](#)
- [Emulation/BackPointers](#)
- [Emulation/DefiningAPI](#)
- [Emulation/Example](#)
- [Emulation/Example/APBDriver](#)
- [Emulation/Example/SPIAgent](#)
- [Emulation/Example/TopLevel](#)

## Topic Overview

A hardware emulation platform, such as Mentor Graphics' Veloce, comprises a solution built upon specialized hardware to deliver high-performance model execution and provide orders of magnitude performance improvement over a traditional software simulator. For Veloce, the specialized hardware includes custom IC technology optimized for the needs of emulation and yet the simple use mode of software simulation.

Since models are compiled into hardware, they must be synthesizable. UVM testbenches rely heavily on object-oriented coding techniques, using SystemVerilog classes and other constructs that are not synthesizable and cannot be executed on an emulator - they must be executed on a software simulator. By combining software simulation for those parts of a testbench that are not synthesizable with hardware emulation for those parts that are synthesizable, you can gain much of the performance benefits of emulation while preserving the power and reuse benefits of UVM-based test environments.

This article describes a co-emulation (a.k.a. co-modeling) approach to writing testbenches that maximizes reuse by enabling truly single-source, fully IEEE 1800 SystemVerilog compliant testbenches that work interchangeably for both simulation and hardware-assisted acceleration. The content provided here is a basic introduction, based predominantly on the Mentor Graphics whitepaper Off to the Races With Your Accelerated SystemVerilog Testbench <sup>[1]</sup>. You should refer to this paper to gain a deeper understanding of the requirements and solutions presented.



## Requirements

Executing a test using co-emulation, where a software simulator runs in tandem with an emulator, presents unique challenges. An emulator is a separate physical device that is executing behavior completely independently from the software simulator. It is connected to the simulator using a *transaction-based* communications link. This has the following implications:

- The synthesizable code that is to be run on the emulator must be completely separated from the non-synthesizable testbench code, placed in separate physical files as well as separate logical hierarchies. These two hierarchies are called the "testbench domain" and the "HDL domain".
- The testbench components that bridge the gap between the test environment and the physical environment - i.e. the so-called transactors like drivers and monitors - must be split into timed and untimed parts, and the parts placed in their respective domains. The untimed portion of each transactor resides in the testbench domain and becomes a proxy to the corresponding timed part in the HDL domain, referred to as BFM. The HDL BFM and its testbench proxy must interact at the transaction level over the emulator-simulator communications link.
- Any timing control statements including #-delays, @-clock-edge synchronizations, and fixed time-interval wait statements left in the testbench environment must be removed, as they impede co-emulation performance. As deemed necessary, these must be remodeled using clock synchronous behavior in the timed HDL domain.
- The link between the simulator and emulator is a physical communications link that deals in packets of data as opposed to transaction object handles. A mapping between packed data packets and transaction objects may therefore be required typically for object-oriented testbenches. Furthermore, in order to achieve highest possible performance the amount of traffic on the physical link must be minimized or optimized.

The above statements can be boiled down to the following three main requirements:

1. Verification code must be completely untimed, i.e. be free of explicit time advance statements. Abstract event synchronizations and waits for abstract events are allowed.

2. Untimed verification code must be separated from timed code into two domains.
3. No cross-domain signal or module references are allowed.

For more detail on these and other requirements, refer to the whitepaper Guidelines for Crafting XLerated SystemVerilog OVCs<sup>[2]</sup> (note, this paper is still OVM-based, but its contents continue to be largely applicable to UVM).

## Methodology Overview

The goal here is to describe a methodology that not only meets the above requirements for co-emulation, but also does so using a single-source codebase, meaning that no code changes are needed to perform tests using either software-only simulation or co-emulation.

The first requirement to address is that there must be two separate logical hierarchies - one to be compiled into the emulator, and one for the testbench environment. This can be accomplished using a "Dual Top" technique. [Click here](#) for a general overview of this technique, and [here](#) for applying the technique to co-emulation.

Once two separate domains are established, the transactors of the testbench must each be split into an untimed transaction-level proxy component in the testbench domain (e.g. a SystemVerilog class like a `uvm_component` derivative) and a synthesizable BFM in the HDL domain (e.g. a SystemVerilog interface). The two parts are to use a collection of user-defined tasks and functions - effectively representing transactions - to implement their required transaction-based communication across the emulator-simulator boundary. An HDL BFM interface implements synthesizable tasks and functions to be called from its proxy via a virtual interface. Conversely, the BFM interface may initiate method calls back into its proxy via an object handle also referred to as back-pointer.

Lastly, since code in the testbench domain must be untimed, any of the aforementioned explicit timing control statements left in that code must be removed, and if necessary re-implemented by synchronization with the timed HDL domain through the task-based BFM communication interface described above.

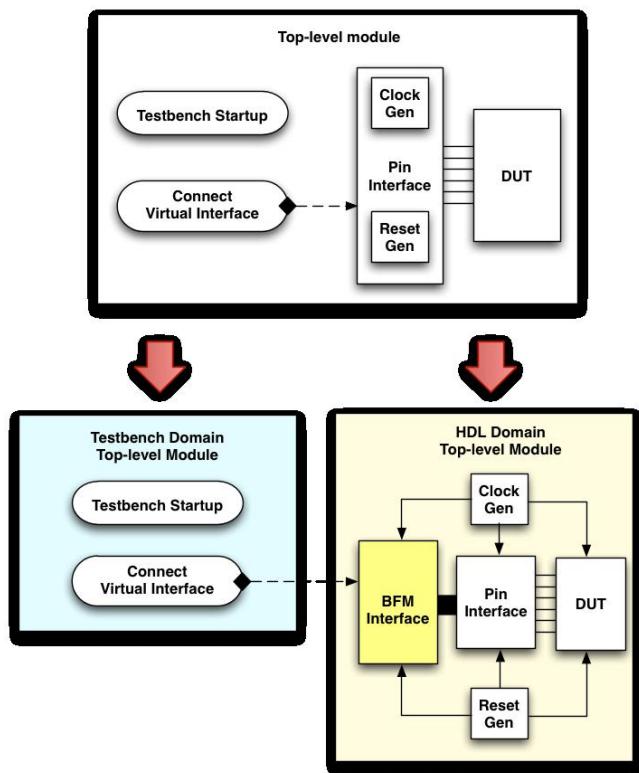
## Worked Example

The methodology pages contain a simple example. Another more comprehensive example illustrating the methodology can be found [here](#).

# Emulation/SeparateTopLevels

## Overview

Co-emulation is done by running two distinct synchronized model evaluations - one on a hardware emulator, and one on a software simulator. These two environments are physically separate and therefore require two module hierarchies, each with its own top-level module in separate files, called the HDL hierarchy and the testbench hierarchy. This dual-root methodology generally follows the procedures detailed in the DualTop article. Additionally, since the accelerated HDL hierarchy will be compiled into the hardware emulator, it must be synthesizable and contain clock-edge synchronized code. The testbench hierarchy contains the remaining verification code. As the testbench code will be executing in a software simulator that must be synchronized with the emulator, the testbench code must be untimed (e.g. "@(posedge clk)", "wait(1ns)", and "#10" are not allowed).



This article will use the bidirectional driver example as a simple testbench to illustrate the required partitioning between synthesizable HDL and untimed testbench top-level modules:

```

module top_tb;

bus_if BUS();
gpio_if GPIO();
bidirect_bus_slave DUT(.bus(BUS), .gpio(GPIO));

// Free running clock
initial
begin
  BUS.clk = 0;
  forever begin
    #10 BUS.clk = ~BUS.clk;
  end
end

// Reset
initial
begin
  BUS.resetn = 0;
  repeat(3) begin
    @(posedge BUS.clk);
  end
  BUS.resetn = 1;
end

// Testbench start up:
initial
begin
  uvm_config_db #(virtual bus_if)::set(null, "uvm_test_top", "BUS_vif" , BUS);
  run_test("bidirect_bus_test");
end

endmodule: top_tb

```

**Testbench Domain****HDL Domain**

Above is the original top-level code, written in a typical single-top manner. In addition to the DUT and interface instantiations, it has a block that generates a clock, a block that generates reset, and a block that initiates the testbench.

The first thing that needs to be done to support co-emulation is to identify the code that belongs in each domain, and move the code accordingly into two separate files, one for the testbench domain and one for the HDL domain. Each file will contain a top-level module. The remainder of this article details the code changes in each domain needed to support this co-emulation methodology.

## HDL Domain

Here is the complete HDL domain top-level code. Details about the code sections follow.

```

module top_hdl;
  bit clk, resetn;

  bus_if BUS(clk, resetn);           Interface and DUT instantiation
  gpio_if GPIO(clk);
  bidirect_bus_slave DUT(.bus(BUS.slave_mp), .gpio(GPIO.out_mp));

  bidirect_bus_driver_bfm DRIVER(.BUS(BUS.master_mp));

  // Free running clock
  // tbx clkgen
  initial
    begin
      clk = 0;
      forever begin
        #10 clk = ~clk;
      end
    end

  // Reset
  // tbx clkgen
  initial
    begin
      resetn = 0;
      #50 resetn = 1;
    end

  endmodule: top_hdl

```

### Clock Gen

### Reset Gen

## DUT and Interface Instantiation

The HDL domain must be written in a synthesizable form, and contains the DUT and any pin interfaces.

```

bus_if BUS(clk, resetn);
gpio_if GPIO(clk);
bidirect_bus_slave DUT(.bus(BUS.slave_mp), .gpio(GPIO.out_mp));

bidirect_bus_driver_bfm DRIVER(.BUS(BUS.master_mp));

```

Additionally, BFM interfaces should be created to contain the timed portions of the driver and monitor transactors.

```

interface bidirect_bus_driver_bfm (bus_if BUS);

  // Synthesizable code implementing the timed transactor portion goes here ...

endinterface

```

TBX currently requires that any interface ports are connected with explicit modports, which define the interface signal directions. If these modports do not exist in the original code, they must be added. If clock and reset signals were generated inside an interface, the associated code must be moved to the top level, and the signals passed into the interface as ports.

```

interface bus_if(input bit clk, resetn);
  // Rest of interface not shown...

  modport master_mp ( ... );

  modport slave_mp ( ... );

endinterface: bus_if

interface gpio_if(input bit clk);
  // Rest of interface not shown...

  modport out_mp( ... );

endinterface: gpio_if

```

### Clock and Reset Generation

In addition to traditional synthesizable content, the HDL side should also contain clock generation and reset generation logic. Note that clock and reset code in conventional testbenches is often behavioral and normally not synthesizable. However, TBX supports synthesis of a superset of RTL code called XRTL (eXtended RTL). According to the XRTL coding guidelines, certain behavioral clock and reset generation blocks are permitted, requiring an extra pragma comment (`// tbx clkgen`). Even though the pragma comment states "clkgen", it should be used for the reset block as well. Clock and reset signals should be generated through non-hierarchical access, so they should be declared at the top level and passed through ports into interfaces. As mentioned above, if the original clock and reset generation code is inside an interface, it must be moved directly to the top-level module.

```

// Free running clock
// tbx clkgen
initial begin
  clk = 0;
  forever begin
    #10 clk = ~clk;
  end
end

```

The statement "repeat (n) @ posedge clock" is not allowed in a TBX clock generation specification block. This must be changed to the equivalent static delay, #50 in this case. Static parameters can be used here as well, for instance imported from a shared parameters package. TBX XRTL also supports variable clock delays, the details of which are not complex but beyond the scope of this article.

```

// Reset
// tbx clkgen
initial begin
  resetn = 0;
  #50 resetn = 1;

```

```
end
```

## Testbench Domain

Here is the complete testbench domain top-level code. Details about the code sections follow.

```
// Top level test bench module
module top_tb;

import uvm_pkg::*;
import bidirect_bus_pkg::*;

// UVM start up:
initial
  begin
    uvm_config_db #(virtual bidirect_bus_driver_bfm)::set(null, "uvm_test_top",
      "top_hdl.DRIVER", top_hdl.DRIVER);
    run_test("bidirect_bus_test");
  end

endmodule: top_tb
```

**Virtual Interface connection**

**Testbench Startup**

### Virtual Interface Connection

In the original testbench, the transactors drove DUT signals through a pin-level interface. TBX does not allow direct access to HDL signals in the testbench. A BFM interface must hence be created in the HDL domain (still a SystemVerilog interface), and the testbench domain communicates with the DUT indirectly through this BFM interface instead of directly through the pin-level interface.

Binding of a virtual interface to a concrete HDL-side interface can be done "natively" in the testbench domain. A cross-reference to the HDL-side interface is then required as shown below, and it is thus important that such binding be done in a top level area in the testbench domain with a global bird's eye view of the entire testbench topology configuration.

```
// Virtual interface handle now points to BFM interface instead of pin-level interface
// NOTE: top_hdl.DRIVER is a hierarchical reference
uvm_config_db #(virtual bidirect_bus_driver_bfm)::set(null, "uvm_test_top", "top_hdl.DRIVER", top_hdl.DRIVER);
```

In order to align with normal DUT connection techniques, TBX also conveniently supports virtual interface binding in the synthesizable HDL domain, right where the corresponding concrete interface is instantiated. It requires the use of a simple pragma to direct TBX not to attempt to synthesize the virtual interface binding code block that must follow the specific format given below.

```
module top_hdl;

  ...

  bidirect_bus_driver_bfm DRIVER(.BUS(BUS.master_mp));

  // tbx vif_binding_block
  initial begin
    import uvm_pkg::uvm_config_db;
    uvm_config_db #(virtual bidirect_bus_driver_bfm)::set(null, "uvm_test_top", $psprintf("%m.DRIVER") , DRIVER);
  end
endmodule
```

```
end

...
endmodule
```

### Testbench Startup

The code that starts the class-based UVM testbench environment remains in the testbench domain.

```
run_test("bidirect_bus_test");
```

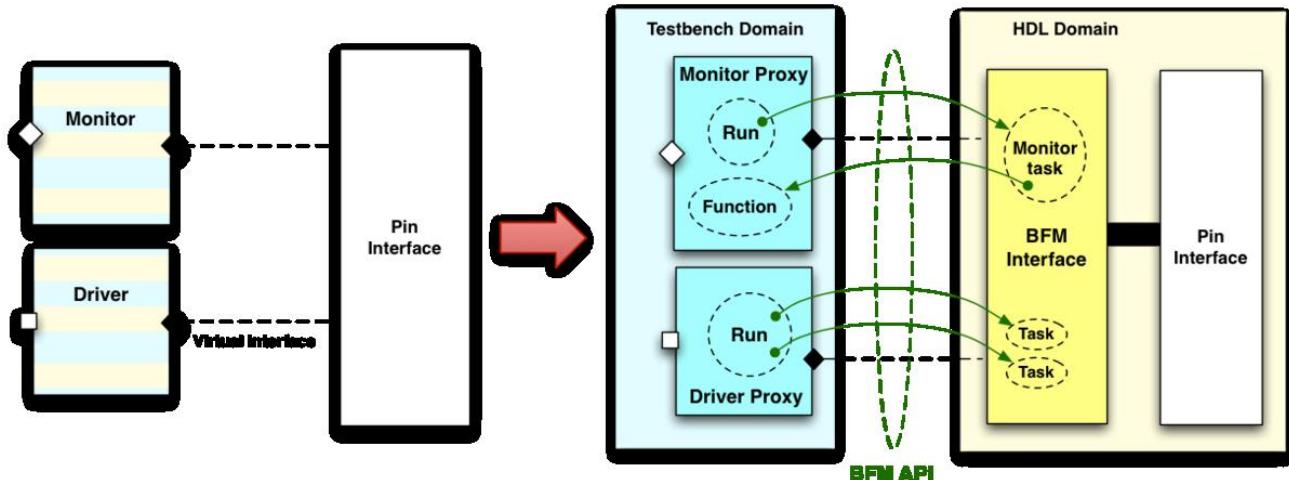
### Shared Parameters

Any parameters that are shared between the testbench domain and the HDL domain should be declared in a shared parameter package imported by both top-level modules. If parameters are used only in the HDL domain, it is not necessary to use a shared parameter package.

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# Emulation/SplitTransactors

## Overview



Driver and monitor transactors contain a mixture of transaction-level code to communicate with the testbench, and clock-driven HDL signal accessing code to communicate with the DUT through a virtual interface. As with the top-level module, you must identify the portions of code that are timed and access signals, and separate that code from the untimed part that manipulates transaction objects.

In general, a transactor follows a common pattern:

1. (Drivers) Get a transaction;
2. (Drivers and Monitors) Interact with DUT signals through a virtual interface;
3. (Drivers and Monitors) Send a transaction out through an analysis port or sequence item port.

The second item, interacting with DUT signals, needs to be separated from the code that sends/receives transactions. This code is to be compiled and executed on the emulator. Data transfer between the emulator and the simulator takes place over a communications link. The communication is implemented as tasks and functions declared in a BFM interface in the HDL domain (still using the SystemVerilog interface construct). The testbench-side transactor object now acts as a proxy for the BFM, calling these BFM tasks and functions (a.k.a. BFM API) to execute interface protocol cycles (i.e. indirectly drive and sample signals), set configuration parameters or get status information. Any data that needs to be exchanged between the BFM and its proxy is passed as task and function input or output arguments.

In addition to the testbench domain initiating activity inside a BFM interface by the BFM proxy calling BFM tasks and functions, it is also possible for the BFM to initiate communication with its proxy by calling non-blocking methods in the proxy through a back-pointer.

## Original Driver Code

The following code, from the bidirectional driver example, shows typical UVM driver code written as a class:

```

class bidirect_bus_driver extends uvm_driver #(bus_seq_item);
virtual bus_if BUS;
task run_phase(uvm_phase phase);
bus_seq_item req;
bus_seq_item rsp;
// Default conditions:
BUS.valid <= 0;
BUS.rnw <= 1;
// Wait for reset to end
@(posedge BUS.resetn);
forever
begin
seq_item_port.get(req); // Start processing req item
repeat(req.delay) begin
@(posedge BUS.clk);
end
BUS.valid <= 1;
BUS.addr <= req.addr;
BUS.rnw <= req.read_not_write;
if(req.read_not_write == 0) begin
BUS.write_data <= req.write_data;
end
while(BUS.ready != 1) begin
@(posedge BUS.clk);
end
// At end of the pin level bus transaction
// Copy response data into the rsp fields:
$cast(rsp, req.clone()); // Clone the req
rsp.set_id_info(req); // Set the rsp id = req id
if(rsp.read_not_write == 1) begin
rsp.read_data = BUS.read_data; // If read - copy returned read data
end
rsp.error = BUS.error; // Copy bus error status
BUS.valid <= 0; // End the pin level bus transaction
seq_item_port.put(rsp); // put returns the response
end
endtask: run_phase
// Rest of driver class not shown...
endclass: bidirect_bus_driver

```

**Testbench Domain**

**HDL Domain**

As with the top-level code this needs to be split into two parts and placed into separate files, where the timed interface protocol is executed on the emulator and the TLM-based portion runs in the software-based UVM simulation environment.

## Testbench Domain Code

The testbench side of the transactor should deal with any TLM-level communication to/from the class-based testbench (e.g. seq\_item\_port and analysis\_port), and any TLM-level manipulation of transaction objects.

```
class bidirect_bus_driver extends uvm_driver #(bus_seq_item);
  virtual bidirect_bus_driver_bfm BFM; // Transactor now uses BFM interface
  // instead of pin interface

  // Constructor, etc. not shown...

  task run_phase(uvm_phase phase);
    bus_seq_item req;
    bus_seq_item rsp;

    // Call BFM task here to wait for reset to end
    BFM.wait_for_reset();

    forever
      begin
        bus_seq_item_s req_s, rsp_s;

        seq_item_port.get(req); // Start processing req item

        // Extract relevant transaction data into packed format:
        bus_seq_item_converter::from_class(req, req_s);

        // Call BFM task here to perform transaction
        // Pass required transaction data to BFM as arguments to the task
        // Get response from BFM from task argument
        BFM.do_item(req_s, rsp_s);

        // Convert from packed format back to transaction object
        bus_seq_item_converter::to_class(rsp, rsp_s);

        rsp.set_id_info(req); // Set the rsp id = req id
        seq_item_port.put(rsp); // put returns the response
      end
    endtask: run_phase
  endclass: bidirect bus driver
```

## Virtual Interface Change

The original transactor performed direct signal access using a handle to the pin interface 'bus\_if'. The updated transactor now performs all timed and bus signal operations through the BFM interface:

```
virtual bidirect_bus_driver_bfm BFM;
```

## Replace Timed Portions with Calls to BFM Tasks and Functions

The timed code that directly accesses signals is moved into tasks in the BFM interface. The updated transactor now calls these tasks through the BFM virtual interface:

```
// Wait for reset to end
BFM.wait_for_reset();

// Call BFM task here to perform transaction
// Pass required transaction data to BFM as arguments to the task
// Get response from BFM from task argument
BFM.do_item(req_s, rsp_s);
```

The mapping between data from the transaction object format ('req' and 'rsp' in the example) and a synthesizable packed format ('req\_s' and 'rsp\_s' in the example) is discussed in the Defining the BFM API article.

## HDL Domain Code

The timed, signal-accessing code of the driver is moved into the BFM interface. This code should be written in a synthesizable style and must adhere to the Veloce TBX SystemVerilog XRTL subset. For instance, XRTL restricts task and function arguments to be input or output - inout and ref arguments are not allowed.

```
interface bidirect_bus_driver_bfm (bus_if BUS);
  // pragma attribute bidirect_bus_driver_bfm partition_interface_xif

  import bidirect_bus_shared_pkg::bus_seq_item_s;

  initial begin
    // Default conditions:
    BUS.valid <= 0;
    BUS.rnw <= 1;
  end

  task wait_for_reset(); // pragma tbx xtf
    @(posedge BUS.resetn);
  endtask

  task do_item(input bus_seq_item_s req, output bus_seq_item_s rsp); // pragma tbx xtf
    @(posedge BUS.clk);
    repeat(req.delay-1) begin
      @(posedge BUS.clk);
    end
    BUS.valid <= 1;
    BUS.addr <= req.addr;
    BUS.rnw <= req.read_not_write;
    if(req.read_not_write == 0) begin
      BUS.write_data <= req.write_data;
    end
    while(BUS.ready != 1) begin
      @(posedge BUS.clk);
    end
    // At end of the pin level bus transaction
    // Copy response data into the rsp fields:
    rsp = req; // Clone the req
    if(req.read_not_write == 1) begin
      rsp.read_data = BUS.read_data; // If read - copy returned read data
    end
    rsp.error = BUS.error; // Copy bus error status
    BUS.valid <= 0; // End the pin level bus transaction
  endtask

endinterface
```

## Pragma Comments

TBX requires special pragma comments for BFM tasks and functions that are called from the proxy in the testbench domain. Firstly, after the interface declaration you must add the following comment:

```
// pragma attribute bidirect_bus_driver_bfm partition_interface_xif
```

Secondly, on the declaration line of the tasks and functions called from the proxy you must add the following comment:

```
// pragma tbx xtf
```

Note that TBX does not permit these BFM API tasks and functions to be called locally within the HDL domain, so be sure to only annotate BFM API routines with this pragma.

## Initial Values and Synchronization

The two default value assignments have been moved into an initial block in the BFM interface. TBX XRTL extends normal RTL synthesis rules by allowing execution of initial blocks:

```
initial begin
  // Default conditions:
  BUS.valid <= 0;
  BUS.rnw <= 1;
```

```
end
```

The line that waits for reset to end is a timing control statement, so it must be moved to a task:

```
task wait_for_reset(); // pragma tbx xtf
  @ (posedge BUS.resetn);
endtask
```

## Main Transaction Code

The original transaction code started with 'repeat (req.delay) begin ...'. TBX XTRL requires that all timed code be clock-synchronous, and in particular time-consuming BFM tasks called from the testbench domain must start with a clock edge synchronization. The original 'repeat' loop has been changed accordingly in the BFM task to the following equivalent code (which is valid since the delay parameter is constrained in the transaction class to be at least 1):

```
@ (posedge BUS.clk);
repeat (req.delay-1) begin
  @ (posedge BUS.clk);
end
```

In the next section, data from the transaction object is applied to the pin interface. The data comes from the input argument to the task, rather than from the transaction object directly. Assignments should be converted to either non-blocking or blocking, following established best coding practices for synthesis.

```
BUS.valid <= 1;
BUS.addr <= req.addr;
BUS.rnw <= req.read_not_write;
if (req.read_not_write == 0) begin
  BUS.write_data <= req.write_data;
end
while (BUS.ready != 1) begin
  @ (posedge BUS.clk);
end
```

## Response Handling

The response data is sent back to the transactor proxy in the testbench domain via the output task argument:

```
// At end of the pin level bus transaction
// Copy response data into the rsp fields:
rsp = req; // Clone the req
if (req.read_not_write == 1) begin
  rsp.read_data = BUS.read_data; // If read - copy returned read data
end
rsp.error = BUS.error; // Copy bus error status
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Emulation/BackPointers

## Overview

In the bidirectional driver example so far, all driver activity is initiated from the testbench domain. Sometimes it is more natural and efficient to initiate activity from the HDL domain, particularly in monitor transactors to send analysis transactions out to the testbench domain, or in pipelined drivers to send back responses. TBX supports "back-pointers", which are simply handles to class-based components in the testbench domain. Class methods can be called from the HDL domain through such back-pointers to initiate communication.

Below is a version of the driver that uses 'get\_next\_item()' and 'item\_done()' to process transactions:

```
class bidirect_bus_driver extends uvm_driver #(bus_seq_item);
bus_seq_item req;
virtual bus_if BUS;

task run_phase(uvm_phase phase);
// Default conditions:
BUS.valid = 0;
BUS.rnw = 1;
// Wait for reset to end
@(posedge BUS.resetn);
forever
begin
  seq_item_port.get_next_item(req); // Start processing req item
  repeat(req.delay) begin
    @(posedge BUS.clk);
  end
  BUS.valid = 1;
  BUS.addr = req.addr;
  BUS.rnw = req.read_not_write;
  if(req.read_not_write == 0) begin
    BUS.write_data = req.write_data;
  end
  while(BUS.ready != 1) begin
    @(posedge BUS.clk);
  end
  // At end of the pin level bus transaction
  // Copy response data into the req fields:
  if(req.read_not_write == 1) begin
    req.read_data = BUS.read_data; // If read - copy returned read data
  end
  req.error = BUS.error; // Copy bus error status
  BUS.valid = 0; // End the pin level bus transaction
  seq_item_port.item_done(); // End of req item
end
endtask: run_phase
endclass: bidirect_bus_driver
```

Testbench Domain

HDL Domain

## Testbench Domain

Timed code must be extracted and moved from the testbench domain to a BFM interface in the HDL domain as before, yet in this case the BFM will be "in control". As a result, the BFM's proxy in the testbench domain just starts up the BFM thread(s) in its 'run()' task, and the transaction-level communication back to the proxy is handled by designated (and appropriately named) methods:

```
class bidirect_bus_driver extends uvm_driver #(bus_seq_item);
  bus_seq_item req;
  virtual bidirect_bus_driver_bfm BFM;
  function void end_of_elaboration_phase(uvm_phase phase);
    BFM.proxy = this;
  endfunction: end_of_elaboration_phase

  task run_phase(uvm_phase phase);
    BFM.run();
  endtask: run_phase

  task try_next_item(output bus_seq_item_s req_s, output bit success);
    seq_item_port.try_next_item(req); // Start processing req item
    success = (req != null);
    if (success)
      bus_seq_item_converter::from_class(req, req_s);
  endtask: try_next_item

  function void item_done(input bus_seq_item_s req_s);
    bus_seq_item req;
    bus_seq_item_converter::to_class(req, req_s);
    this.req.copy(req);
    seq_item_port.item_done(); // End of req item
  endfunction: item_done
endclass: bidirect_bus_driver
```

## Setting the Back-Pointer

Somewhere after binding the BFM virtual interface but before the start of the UVM run phase, the BFM's handle back to its proxy must be assigned. For this example it must be between the connect and run phases, i.e. in the end-of-elaboration phase:

```
function void end_of_elaboration_phase(uvm_phase phase);
  BFM.proxy = this;
endfunction: end_of_elaboration_phase
```

## Start Transaction Processing

Here, the main flow of control is based in the BFM, so the BFM's proxy merely initiates and then yields to the BFM:

```
task run_phase(uvm_phase phase);  
    BFM.run();  
endtask: run_phase
```

## Defining Back-Pointer Methods

The untimed behavior of the driver in the testbench domain that manipulates class-based transactions is moved into functions or zero-time tasks to be called by the driver BFM. The method 'try\_next\_item()' of the UVM 'seq\_item\_port', even though it is zero-time, is technically a task, so it must be wrapped in a task rather than a function. Since the BFM will need to know if the call actually provided an item, a success status bit is further added:

```

task try_next_item(output bus_seq_item_s req_s, output bit success);
    seq_item_port.try_next_item(req); // Start processing req item
    success = (req != null);
    if (success)
        bus_seq_item_converter::from_class(req, req_s);
endtask: try_next_item

function void item_done(input bus_seq_item_s req_s);
    bus_seq_item req;
    bus_seq_item_converter::to_class(req, req_s);
    this.req.copy(req);
    seq_item_port.item_done(); // End of req item
endfunction: item_done

```

## HDL Domain

## Declaring the Back-Pointer

Back-pointers to class-based components in the testbench domain are declared inside BFM interfaces in the HDL domain. A pragma comment is not required, but the one specified in the example code can be seen as a kind of TBX compiler directive to enable one-way caller performance optimization. Effectively, the comment tells TBX that the emulator can keep running and that the emulator clocks need not be stopped for communication/synchronization with the

simulator and to yield to testbench threads at this time. This is feasible because the method 'item\_done()' is a one-way function - a void function without output arguments and without side effects (like a 'pure' void function in C/C++).

```
bidirect_bus_driver proxy; // pragma tbx oneway proxy.item_done
```

### Main Run Thread

Back-pointers are most helpful when the bulk of processing is done on the HDL side, making it more natural for this side to be an initiator. This pattern works particularly well for monitor transactors. In this example of the bidirectional driver, the main forever loop is implemented on the HDL side, and it communicates with the testbench side to get a request transaction and send back the response:

```
task run_phase(uvm_phase phase); // pragma tbx xtf
  // Other code not shown...

  forever begin
    bit success;

    proxy.try_next_item(req, success); // Start processing req item

    while (!success) begin
      @(posedge BUS.clk);
      proxy.try_next_item(req, success); // Start processing req item
    end

    // Process transaction, get response ...

    proxy.item_done(req); // End of req item
  end
endtask
```

( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

# Emulation/DefiningAPI

## Overview

The timed portion of the original transactor code must be moved over to the HDL domain. This means that the pertinent lines of code must be extracted and rearranged into synthesizable code blocks inside a BFM interface, a SystemVerilog interface, involving in particular the creation of tasks and functions in that interface to effectively define a transaction-based API for the BFM, accessible from the testbench domain. When designing the BFM API, consider that since the two domains are running on separate hardware platforms, workstation and emulator, communication between them is rather expensive. Consequently, communication should be kept relatively low-frequency and data-rich, as opposed to high frequency and data-poor (e.g. imagine a Ferrari driving through a large city's downtown intersections, hitting every red light).

The data required in the HDL domain to drive and monitor the DUT's pin interfaces typically stems from stimulus and analysis transactions in the testbench domain. Transactions are generally class-based and model an abstraction of the pin-level DUT data. They are optimized for class-based communication and manipulation. The HDL domain does not support class objects, however, and hence transaction data from the testbench domain may need to be converted into synthesizable DUT-compatible data and vice-versa.

## BFM Task and Function Arguments

When the signal-accessing portion of transactor code is moved to a BFM interface, transaction data needs to be passed between the BFM and its proxy via synthesizable task arguments. Stimulus data enters the HDL BFM from the testbench domain and analysis data exits the BFM to the testbench domain. The BFM implementation ultimately determines which pieces of transaction data are manipulated in the HDL domain and must be communicated via the BFM task and function API.

Given below is the main BFM task for the bidirectional driver example, declared inside the driver BFM interface in the HDL domain and called from the driver proxy in the testbench domain:

```
task do_item(input bus_seq_item_s req, output bus_seq_item_s rsp); // pragma tbx xtf
  @(posedge BUS.clk);
  repeat (req.delay-1) begin
    @(posedge BUS.clk);
  end

  BUS.valid <= 1;
  BUS.addr <= req.addr;
  BUS.rnw <= req.read_not_write;
  if(req.read_not_write == 0) begin
    BUS.write_data <= req.write_data;
  end

  while(BUS.ready != 1) begin
    @(posedge BUS.clk);
  end
endtask
```

```

// At end of the pin level bus transaction
// Copy response data into the rsp fields:
rsp = req; // Clone the req
if(req.read_not_write == 1) begin
    rsp.read_data = BUS.read_data; // If read - copy returned read data
end
rsp.error = BUS.error; // Copy bus error status

BUS.valid <= 0; // End the pin level bus transaction
endtask

```

This task needs from the transaction object the fields 'addr', 'read\_not\_write', 'write\_data', and 'delay' in case of a request, and the fields 'error' and 'read\_data' in case of a response transaction.

If there are just a few data items, such as address and data, which are already in a packed format, the data is readily passed directly as individual arguments to the task. On the other hand, if there are many relevant transaction data fields, it may be more convenient to define a synthesizable packed struct container and pass all the data via a single struct task argument. For the current driver example, the BFM requires four request data fields and two response data fields, so a struct is defined:

```

package bidirect_bus_pkg;

typedef struct packed {
    // Request fields
    logic[31:0] addr;
    logic[31:0] write_data;
    bit read_not_write;
    int delay;

    // Response fields
    bit error;
    logic[31:0] read_data;
} bus_seq_item_s;

// Rest of package not shown...

endpackage: bidirect_bus_pkg

```

The BFM task 'do\_item(...)' uses this packed struct type to pass data to/from the proxy. Note that TBX does not support inout and ref arguments, only input and output arguments. So, the task arguments look like:

```
task do_item(input bus_seq_item_s req, output bus_seq_item_s rsp);
```

### Converting Between Testbench Transactions and HDL BFM Data

Mapping between higher level transaction objects and low level pin wiggles - sometimes referred to as data packing/unpacking - is conventionally done either directly as part of a testbench component's run\_phase() task, or indirectly by calling helper functions from that run\_phase() task. For the split transactors it can still be done this way, i.e. in a BFM's proxy, or it can be delegated to an external converter class as follows:

```
// Delegate class that converts between bus_seq_item transaction and
// synthesizable packed struct

class bus_seq_item_converter;

  static function void from_class(input bus_seq_item t, output bus_seq_item_s s);
    s.addr = t.addr;
    s.write_data = t.write_data;
    s.read_not_write = t.read_not_write;
    s.delay = t.delay;
    s.error = t.error;
    s.read_data = t.read_data;
  endfunction

  static function void to_class(output bus_seq_item t, input bus_seq_item_s s);
    t = new();
    t.addr = s.addr;
    t.write_data = s.write_data;
    t.read_not_write = s.read_not_write;
    t.delay = s.delay;
    t.error = s.error;
    t.read_data = s.read_data;
  endfunction

endclass: bus_seq_item_converter
```

This class is used to convert between the original bus sequence item object and the similar struct defined earlier:

```
bus_seq_item_converter::from_class(req, req_s);
BFM.do_item(req_s, rsp_s);
bus_seq_item_converter::to_class(rsp, rsp_s);
```

# Emulation/Example

## Overview

This page steps through the process of converting a comprehensive UVM example testbench to be ready for co-emulation with Veloce TBX. It follows the methodology introduced on this page, resulting in a single-source testbench that is reusable for either hardware-assisted acceleration or conventional software simulation.

## Split Top Level

The first thing to be done is to split the single top level in the original code into two top-level domains, one for the DUT and other accelerated HDL code, and one for the testbench.

[Click here](#) to see the original top-level code and the two domain top-levels.

Most of the code in the original top level should be placed in the HDL top-level module, with the exception of the UVM initial block. That block should be moved to a separate top-level module.

### The HDL Domain

The DUT domain top-level module should contain:

- DUT instantiation
- Any DUT pin interface instantiations
- Clock and reset signal generation
- BFM interface instantiations

### Interface Declarations and Instantiations

TBX currently requires that any pin interface bundles passed as ports must use a modport. Here, you would instantiate the transactor BFM interfaces and connect the DUT pin interface with a modport.

```
apb_driver_bfm  APB_DRIVER(APB.driver_mp);  
apb_monitor_bfm APB_MONITOR(APB.monitor_mp);  
  
spi_driver_bfm  SPI_DRIVER(SPI.driver_mp);  
spi_monitor_bfm SPI_MONITOR(SPI.monitor_mp);
```

In this example, the DUT is connected via signal-level ports, so no modport is necessary for that connection. If the DUT were connected via a pin interface bundle, then a modport would be needed.

### Clock and Reset Generation

TBX does not allow the clock generation to be inside an interface. You should move any clock or reset generation out of an interface and into the HDL top-level module, and pass them into interfaces through ports. In this example, the clock and reset were already in the top-level module, so no changes were necessary in that regard.

However, TBX requires that clock and reset generation be in separate initial blocks, and you must annotate the clock and reset generation code with a pragma comment:

```
// tbx clkgen  
initial begin
```

```

PCLK = 0;
forever begin
  #10ns PCLK = ~PCLK;
end
end

// tbx clkgen
initial begin
  PRESETn = 0;
  #80 PRESETn = 1;
end

```

### Virtual Interface Binding

Since TBX supports HDL-side virtual interface binding, the HDL domain may also contain the code to put the virtual interface handles to the various BFM proxies into the UVM configuration database to set up the communication mechanism with the testbench domain.

```

// tbx vif_binding_block
initial begin
  import uvm_pkg::uvm_config_db;
  uvm_config_db #(virtual apb_driver_bfm)::set(null, "uvm_test_top", $psprintf("%m.APB_DRIVER"), APB_DRIVER);
  ...
end

```

### The Testbench Domain

The testbench domain contains the code to instantiate the test object and start the test environment phasing. And it may also handle the virtual interface binding instead of doing this on the HDL side as above (but not on both sides).

```

initial begin
  uvm_config_db #(virtual apb_driver_bfm)::set(null, "uvm_test_top", "top_hdl.APB_DRIVER", top_hdl.APB_DRIVER);
  ...
  run_test();
end

```

### Split Transactors

In this example, there are two interfaces: an APB interface and an SPI interface. Each interface has a driver and a monitor transactor. These four transactors need to be split into a TLM-level portion for the testbench domain and an acceleratable BFM for the HDL domain. In each BFM task or function, all assignments to signals are converted into appropriate blocking or non-blocking assignments. In this example, the BFM tasks lend themselves to be written in an implicit state machine style.

## Clock Synchronization

Note that all tasks in the APB and SPI BFM must start by synchronizing to a clock edge. This is required by TBX XRTL.

## Communication Format

The two parts of the split transactor communicate through a task call via a virtual interface handle to the BFM interface. TBX requires that the data format of the task arguments be in a packable format. When there is a lot of data to be transferred, a delegate class to perform the conversion between transaction class data and the BFM packable data is useful:

### APB:

```
class apb_seq_item_converter;

    static function void from_class(input apb_seq_item t, output apb_seq_item_vector_t v);
        apb_seq_item_s s;
        s.addr = t.addr;
        s.data = t.data;
        s.we = t.we;
        s.delay = t.delay;
        s.error = t.error;
        v = s;
    endfunction

    static function void to_class(output apb_seq_item t, input apb_seq_item_vector_t v);
        apb_seq_item_s s;
        s = v;
        t = new();
        t.addr = s.addr;
        t.data = s.data;
        t.we = s.we;
        t.delay = s.delay;
        t.error = s.error;
    endfunction

endclass: apb_seq_item_converter
```

### SPI:

For the SPI, there is not very much data to be transferred, so no conversion class is necessary. The data is passed directly as arguments to the BFM task.

### APB Agent

Click here to see the original APB agent code and the split transactors for the driver and monitor.

Splitting these transactors is relatively straightforward, by putting the time-consuming, signal-accessing code into tasks in the BFM.

### APB Driver

All of the signal-accessing code has been moved to a task named do\_item() on the BFM. The driver calls this task and the transactor data is translated using the delegate class:

```
apb_seq_item_converter::from_class(req, req_s);
BFM.do_item(req_s, psel_index, rsp_s);
apb_seq_item_converter::to_class(rsp, rsp_s);
```

The code in the original driver that ran once at the beginning of the run task has been placed in an initial block in the BFM:

```
initial begin
    APB.PSEL <= 0;
    APB.PENABLE <= 0;
    APB.PADDR <= 0;
end
```

### APB Monitor

The monitor uses a back-pointer to send the transaction data back to the testbench proxy. The testbench monitor sets the value of the back-pointer in the end\_of\_elaboration phase:

```
function void apb_monitor::end_of_elaboration_phase(uvm_phase phase);
    BFM.proxy = this;
endfunction: end_of_elaboration_phase
```

The testbench monitor proxy just starts the BFM task named run() and defines the write() function that will be called from the BFM:

```
task apb_monitor::run_phase(uvm_phase phase);
    BFM.run(apb_index);
endtask: run_phase

function void apb_monitor::write(apb_seq_item_s item_s);
    apb_seq_item item;

    apb_seq_item_converter::to_class(item, item_s);
    this.item.copy(item);
    ap.write(this.item);
```

```
endfunction: write
```

In the BFM task, a back pointer is declared with a special TBX pragma comment. When a transaction is recognized in the BFM's run() task, it is sent back to the testbench by calling a function in the transactor through the back-pointer:

```
apb_monitor proxy; // pragma tbx oneway proxy.write

task run(int index); // pragma tbx xtf

forever begin
    // Detect the protocol event on the TBAI virtual interface

    ...

    // Publish the item to the subscribers
    proxy.write(item);
end
endtask: run
```

## SPI Agent

Click here to see the original SPI driver and monitor, along with the split transactors.

The SPI transactors pose a more significant challenge to split and maintain equivalent behavior. The challenge is not with the methodology, however, but is with the process of writing the transactor behavior in an equivalent but synthesizable manner.

The original SPI transactors use an interface-specific clock, SPI.clk. The code is written to be sensitive to both edges of this clock. In order to be synthesizable yet equivalent, you have to bring in the system clock and make all activity on a single edge of that clock.

## SPI Driver

The original driver starts out with code that is not allowed by the methodology rules (no timing control in the testbench side), but is also not synthesizable:

```
while(SPI.cs === 8'hxx) begin
    #1;
end
```

The HDL domain driver defines an init() task to implement this in a synthesizable way:

```
task init(); // pragma tbx xtf
    @(negedge SPI.PCLK);
    while(SPI.cs != 8'hff) @(negedge SPI.PCLK);
endtask: init
```

A part of the original driver code that is sensitive to both edges of the SPI clock:

```
if(req.RX_NEG == 1) begin
    @(posedge SPI.clk);
```

```

end
else begin
  @(negedge SPI.clk);
end

```

This needs to be rewritten in order to be synthesizable by using the common edge of the system clock and using a task argument RX\_NEG:

```

for(int i = 1; i < num_bits-1; i++) begin
  @(negedge SPI.PCLK);           //--
  while(SPI.clk == RX_NEG) @(negedge SPI.PCLK); //  |- mimics if (RX_NEG == 1) @(posedge SPI.clk) else @(negedge SPI.clk)
  while(SPI.clk != RX_NEG) @(negedge SPI.PCLK); //--
  SPI.miso <= spi_data[i];
  if(SPI.cs == 8'hff) begin
    break;
  end
end

```

## SPI Monitor

The SPI monitor is implemented in a similar way as the APB monitor. A main run() task is implemented in the BFM interface and when a transaction is recognized, the BFM calls a write() function in the testbench monitor through a back-pointer. SPI clock handling is done in the same was as in the SPI driver, by bringing in the system clock.

The main challenge of the monitor is how to handle the fork/join\_any in the original code:

```

fork
begin
  while(SPI.cs != 8'hff) begin
    @(SPI.clk);
    if(SPI.clk == 1) begin
      item.nedge_mosi[p] = SPI.mosi;
      item.nedge_miso[p] = SPI.miso;
      p++;
    end
    else begin
      item.pedge_mosi[n] = SPI.mosi;
      item.pedge_miso[n] = SPI.miso;
      n++;
    end
  end
end
begin
  @(SPI.clk);
  @(SPI.cs);
end
join_any

```

```
 disable fork;
```

There are two processes that are forked, one that collects bits from the serial stream and one that watches the chip-select signal. Because of the join\_any followed by disable fork, if the chip-select changes value at any time during the bit collection, the bit collection is aborted. Also, the sampling of the chip-select is done on both edges of the SPI clock.

This behavior is implemented in a synthesizable way in the BFM:

```
clk_val = SPI.clk;                                // --
@(negedge SPI.PCLK);                                //  |- mimics @(SPI.clk);
while(SPI.clk == clk_val)  @(negedge SPI.PCLK); // --

while(SPI.cs == cs) begin
  if(SPI.clk == 1) begin
    nedge_mosi[p] <= SPI.mosi;
    nedge_miso[p] <= SPI.miso;
    p++;
  end
  else begin
    pedge_mosi[n] <= SPI.mosi;
    pedge_miso[n] <= SPI.miso;
    n++;
  end
  clk_val = SPI.clk;                                // ---
  @(negedge SPI.PCLK);                                //  |
  while(SPI.clk == clk_val) begin //  |- mimics @(SPI.clk) with premature break on SPI.cs change
    @(negedge SPI.PCLK);                                //  |
    if (SPI.cs != cs) break;                                // ---
  end
end
```

## Remove Timing from Testbench Domain

The main\_phase() task of the original SPI test class is as follows:

```
task spi_test::main_phase(uvm_phase phase);
  send_spi_char_seq spi_char_seq = send_spi_char_seq::type_id::create("spi_char_seq");

  phase.raise_objection(this, "Starting spi_char_seq");
  spi_char_seq.start(m_env.m_v_sqr.apb);
  #100ns;
  phase.drop_objection(this, "Finished spi_char_seq");
endtask: main_phase
```

This code has a pound-delay timing control statement that violates the methodology rules. To handle any such delays in the test or in sequences, you can define a general-purpose delay task in an HDL-side interface - in this example, the task is in the INTR interface:

```
task wait_n_cycles(int n); // pragma tbx xtf
  @(posedge PCLK);
  assert(n>0);
  repeat (n-1) @(posedge PCLK);
endtask: wait_n_cycles
```

To call this task from any point in the testbench, you would need global access to the virtual interface handle. Using the technique described here, a "Wait\_for\_interface\_signal" task can be defined, which can be called from places in the testbench other than the transactors:

```
// This task is a convenience method placed in the configuration object
// for sequences and tests waiting for time to elapse

task spi_env_config::pound_delay(int n);
  if(n == 0) begin
    `ovm_error("SPI_ENV_CONFIG:",
      $sformatf("Argument n for pound_delay must be greater than zero"))
  end
  if (n % 20 == 0) begin
    INTR.wait_n_cycles(n);
  end
  else begin
    `ovm_warning("SPI_ENV_CONFIG:",
      $sformatf("Argument n=%0d for pound_delay not a multiple of 20; delay rounded up to next integer multiple %0d", n, (n/20+1)*20))
    INTR.wait_n_cycles(n/20+1);
  end
endtask: pound_delay
```

Now, the test can be rewritten to use the configuration task instead of using the pound-delay statement:

```
task spi_test::main_phase(uvm_phase phase);
  send_spi_char_seq spi_char_seq = send_spi_char_seq::type_id::create("spi_char_seq");

  phase.raise_objection(this, "Starting spi_char_seq");
  spi_char_seq.start(m_env.m_v_sqr.apb);
  m_env_cfg.pound_delay(100); // ----- Call to "wait_for_interface_signal" convenience method
  phase.drop_objection(this, "Finished spi_char_seq");
endtask: main_phase
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# Emulation/Example/APBDriver

## APB Driver

Here is the original run task for the APB driver:

```
task apb_driver::run_phase(uvm_phase phase);
  apb_seq_item req;
  apb_seq_item rsp;
  int psel_index;

  forever begin
    APB.PSEL <= 0;
    APB.PENABLE <= 0;
    APB.PADDR <= 0;
    seq_item_port.get_next_item(req);
    repeat(req.delay)
      @(posedge APB.PCLK);
    psel_index = sel_lookup(req.addr);
    if(psel_index >= 0) begin
      APB.PSEL[psel_index] <= 1;
      APB.PADDR <= req.addr;
      APB.PWDATA <= req.data;
      APB.PWRITE <= req.we;
      @(posedge APB.PCLK);
      APB.PENABLE <= 1;
      while (!APB.PREADY)
        @(posedge APB.PCLK);
      if(APB.PWRITE == 0) begin
        req.data = APB.PRDATA;
      end
    end
    else begin
      `ovm_error("RUN", $sformatf("Access to addr %0h out of APB address range", req.addr))
      req.error = 1;
    end
    seq_item_port.item_done();
  end

  endtask: run_phase
```

### Testbench Domain APB Driver Proxy

Here is the code for the testbench portion of the transactor:

```
task apb_driver::run_phase(uvm_phase phase);
  apb_seq_item req;
  apb_seq_item rsp;
  int psel_index;

  forever begin
    apb_seq_item_s req_s, rsp_s;

    seq_item_port.get_next_item(req);
    psel_index = sel_lookup(req.addr);
    if(psel_index < 0) begin
      `ovm_error("RUN", $sformatf("Access to addr %0h out of APB address range", req.addr))
    end
    apb_seq_item_converter::from_class(req, req_s);
    BFM.do_item(req_s, psel_index, rsp_s);
    apb_seq_item_converter::to_class(rsp, rsp_s);
    req.copy(rsp);
    seq_item_port.item_done();
  end
endtask: run_phase
```

### HDL Domain APB Driver BFM

```
interface apb_driver_bfm (apb_if APB);
  // pragma attribute apb_driver_bfm partition_interface_xif

  import apb_shared_pkg::apb_seq_item_s;

  initial begin
    APB.PSEL <= 0;
    APB.PENABLE <= 0;
    APB.PADDR <= 0;
  end

  task do_item(apb_seq_item_s req, int psel_index, output apb_seq_item_s rsp); // pragma tbx xtf
    @(posedge APB.PCLK);
    repeat(req.delay-1) // ok since delay is constrained to be between 1 and 20
      @(posedge APB.PCLK);

    rsp = req;
    rsp.error = (psel_index < 0);
```

```

if(rsp.error)  return;

APB.PSEL[psel_index] <= 1;
APB.PADDR <= req.addr;
APB.PWDATA <= req.data;
APB.PWRITE <= req.we;
@(posedge APB.PCLK);
APB.PENABLE <= 1;
while (!APB.PREADY)
  @(posedge APB.PCLK);
if(APB.PWRITE == 0) begin
  rsp.data = APB.PRDATA;
end
APB.PSEL <= 0;
APB.PENABLE <= 0;
APB.PADDR <= 0;
endtask: do_item

endinterface: apb_driver_bfm

```

## APB Monitor

Here is the original run task for the APB monitor:

```

task apb_monitor::run_phase(uvm_phase phase);
  apb_seq_item item;
  apb_seq_item cloned_item;

  item = apb_seq_item::type_id::create("item");

  forever begin
    // Detect the protocol event on the TBAI virtual interface
    @(posedge APB.PCLK);
    if(APB.PREADY && APB.PSEL[apb_index]) begin
      // Assign the relevant values to the analysis item fields
      item.addr = APB.PADDR;
      item.we = APB.PWRITE;
      if(APB.PWRITE) begin
        item.data = APB.PWDATA;
      end
      else begin
        item.data = APB.PRDATA;
      end
      // Clone and publish the cloned item to the subscribers
      $cast(cloned_item, item.clone());
    end
  end

```

```

    ap.write(cloned_item);
  end
end
endtask: run_phase

```

### Testbench Domain APB Monitor Proxy

```

function void apb_monitor::end_of_elaboration_phase(uvm_phase phase);
  BFM.proxy = this;
endfunction: end_of_elaboration_phase

task apb_monitor::run_phase(uvm_phase phase);
  BFM.run(apb_index);
endtask: run_phase

function void apb_monitor::write(apb_seq_item_s item_s);
  apb_seq_item item;
  apb_seq_item_converter::to_class(item, item_s);
  this.item.copy(item);
  ap.write(this.item);
endfunction: write

```

### HDL Domain APB Monitor BFM

```

interface apb_monitor_bfm(apb_if APB);
  // pragma attribute apb_monitor_bfm partition_interface_xif

  import apb_shared_pkg::apb_seq_item_s;
  import apb_agent_pkg::apb_monitor;

  apb_monitor proxy; // pragma tbx oneway proxy.write

  task run(int index); // pragma tbx xtf
    apb_seq_item_s item;
    @(posedge APB.PCLK);

    forever begin
      // Detect the protocol event on the TBAI virtual interface
      @(posedge APB.PCLK);
      if(APB.PREADY & APB.PSEL[index]) begin // index identifies PSEL line this monitor is connected to
        // Assign the relevant values to the analysis item fields
        item.addr = APB.PADDR;
        item.we = APB.PWRITE;
        if(APB.PWRITE) begin

```

```
    item.data = APB.PWDATA;
  end
  else begin
    item.data = APB.PRDATA;
  end
  // Publish the item to the subscribers
  proxy.write(item);
end
endtask: run

endinterface: apb_monitor_bfm
```

# Emulation/Example/SPIAgent

## Original SPI Driver

```
// This driver is really a SPI slave responder
task spi_driver::run_phase(uvm_phase phase);
    spi_seq_item req;
    spi_seq_item rsp;
    int no_bits;

    SPI.miso = 1;
    while(SPI.cs === 8'hxx) begin
        #1;
    end

    forever begin
        seq_item_port.get_next_item(req);
        while(SPI.cs == 8'hff) begin
            @(SPI.cs);
        end
        no_bits = req.no_bits;
        if(no_bits == 0) begin
            no_bits = 128;
        end
        SPI.miso = req.spi_data[0];
        for(int i = 1; i < no_bits-1; i++) begin
            if(req.RX_NEG == 1) begin
                @(posedge SPI.clk);
            end
            else begin
                @(negedge SPI.clk);
            end
            SPI.miso = req.spi_data[i];
            if(SPI.cs == 8'hff) begin
                break;
            end
        end
        seq_item_port.item_done();
    end
endtask: run_phase
```

### Testbench Domain SPI Driver Proxy

```
// This driver is really a SPI slave responder
task spi_driver::run_phase(uvm_phase phase);
    spi_seq_item req;
    spi_seq_item rsp;

    BFM.init();

    forever begin
        seq_item_port.get_next_item(req);
        BFM.do_item(req.spi_data, req.no_bits, req.RX_NEG);
        seq_item_port.item_done();
    end
endtask: run_phase
```

### HDL Domain SPI Driver BFM

```
interface spi_driver_bfm (spi_if SPI);
    // pragma attribute spi_driver_bfm partition_interface_xif

    initial begin
        SPI.miso = 1;
    end

    task init(); // pragma tbx xtf
        @(negedge SPI.PCLK);
        while(SPI.cs != 8'hff) @(negedge SPI.PCLK);
    endtask: init

    // This driver is really an SPI slave responder
    task do_item(logic[127:0] spi_data, bit[6:0] no_bits, bit RX_NEG); // pragma tbx xtf
        bit[7:0] num_bits;

        @(negedge SPI.PCLK);

        while(SPI.cs == 8'hff) @(negedge SPI.PCLK);

        num_bits = no_bits;
        if(num_bits == 0) begin
            num_bits = 128;
        end
        SPI.miso <= spi_data[0];
        for(int i = 1; i < num_bits-1; i++) begin
            @(negedge SPI.PCLK); // --| mimics:
            while(SPI.clk == RX_NEG) @(negedge SPI.PCLK); // | if (RX_NEG == 1)
        end
    endtask: do_item
endinterface: spi_driver_bfm
```

```

while(SPI.clk != RX_NEG) @(negedge SPI.PCLK); // --| @(posedge SPI.clk) else @(negedge SPI.clk)
  SPI.miso <= spi_data[i];
  if(SPI.cs == 8'hff) begin
    break;
  end
end
endtask: do_item

endinterface: spi_driver_bfm

```

## Original SPI Monitor

```

task spi_monitor::run_phase(uvm_phase);
  spi_seq_item item;
  spi_seq_item cloned_item;
  int n;
  int p;

  item = spi_seq_item::type_id::create("item");

  while(SPI.cs === 8'hxx) begin
    #1;
  end

  forever begin

    while(SPI.cs === 8'hff) begin
      @(SPI.cs);
    end

    n = 0;
    p = 0;
    item.nedge_mosi = 0;
    item.pedge_mosi = 0;
    item.nedge_miso = 0;
    item.pedge_miso = 0;
    item.cs = SPI.cs;

    fork
      begin
        while(SPI.cs != 8'hff) begin
          @(SPI.clk);
          if(SPI.clk == 1) begin
            item.nedge_mosi[p] = SPI.mosi;
            item.nedge_miso[p] = SPI.miso;

```

```

        p++;
    end
    else begin
        item.pedge_mosi[n] = SPI.mosi;
        item.pedge_miso[n] = SPI.miso;
        n++;
    end
    end
begin
    @(SPI.clk);
    @(SPI.cs);
end
join_any
disable fork;

// Clone and publish the cloned item to the subscribers
$cast(cloned_item, item.clone());
ap.write(cloned_item);
end
endtask: run_phase

```

### Testbench Domain SPI Monitor Proxy

```

function void spi_monitor::end_of_elaboration_phase(uvm_phase phase);
    BFM.proxy = this;
endfunction: end_of_elaboration_phase

task spi_monitor::run_phase(uvm_phase);
    item = spi_seq_item::type_id::create("item");
    BFM.run();
endtask: run_phase

function void spi_monitor::write(logic[127:0] nedge_mosi, pedge_mosi, nedge_miso, pedge_miso, logic[7:0] cs);
    spi_seq_item cloned_item;

    item.nedge_mosi = nedge_mosi;
    item.pedge_mosi = pedge_mosi;
    item.nedge_miso = nedge_miso;
    item.pedge_miso = pedge_miso;
    item.cs = cs;
    // Clone and publish the cloned item to the subscribers
    $cast(cloned_item, item.clone());
    ap.write(cloned_item);
endfunction: write

```

### HDL Domain SPI Monitor BFM

```

interface spi_monitor_bfm(spi_if SPI);
// pragma attribute spi_monitor_bfm partition_interface_xif

import spi_agent_pkg::spi_monitor;

spi_monitor proxy; // pragma tbx oneway proxy.write

task run(); // pragma tbx xtf
  logic[127:0] nedge_mosi;
  logic[127:0] pedge_mosi;
  logic[127:0] nedge_miso;
  logic[127:0] pedge_miso;
  logic[7:0] cs;
  int n;
  int p;
  bit clk_val;

  @(negedge SPI.PCLK);

  while(SPI.cs != 8'hff) @(negedge SPI.PCLK);

  forever begin
    while(SPI.cs == 8'hff) @(negedge SPI.PCLK);

    n = 0;
    p = 0;
    nedge_mosi <= 0;
    pedge_mosi <= 0;
    nedge_miso <= 0;
    pedge_miso <= 0;
    cs <= SPI.cs;

    clk_val = SPI.clk; // --
    @(negedge SPI.PCLK); // -- | - mimics @(SPI.clk);
    while(SPI.clk == clk_val) @(negedge SPI.PCLK); // --

    while(SPI.cs == cs) begin
      if(SPI.clk == 1) begin
        nedge_mosi[p] <= SPI.mosi;
        nedge_miso[p] <= SPI.miso;
        p++;
      end
      else begin

```

```
pedge_mosi[n] <= SPI.mosi;
pedge_miso[n] <= SPI.miso;
n++;
end
clk_val = SPI.clk;           // ---
@(negedge SPI.PCLK);         //   |
while(SPI.clk == clk_val) begin //   |- mimics @(SPI.clk) with premature break on SPI.cs change
  @(negedge SPI.PCLK);       //   |
  if (SPI.cs != cs) break;   // ---
end
end

// Publish to the subscribers
proxy.write(nedge_mosi, pedge_mosi, nedge_miso, pedge_miso, cs);
end
endtask: run

endinterface: spi_monitor_bfm
```

# Emulation/Example/TopLevel

## Original Single Top Level

```
module top_tb;

import uvm_pkg::*;
import spi_test_lib_pkg::*;

// PCLK and PRESETn
//
logic PCLK;
logic PRESETn;

//
// Instantiate the interfaces:
//
apb_if APB(PCLK, PRESETn); // APB interface
spi_if SPI();           // SPI Interface
intr_if INTR();          // Interrupt

//
// DUT
spi_top DUT(
    // DUT pin connections not shown...
);

// UVM initial block:
// Virtual interface wrapping & run_test()
initial begin
    uvm_config_db #(virtual apb_if)::set(null, "uvm_test_top", "APB_vif" , APB);
    uvm_config_db #(virtual spi_if)::set(null, "uvm_test_top", "SPI_vif" , SPI);
    uvm_config_db #(virtual intr_if)::set(null, "uvm_test_top", "INTR_vif", INTR);
    run_test();
end

//
// Clock and reset initial block:
//
initial begin
    PCLK = 0;
    PRESETn = 0;
    repeat(8) begin
        #10ns PCLK = ~PCLK;
    end
end
```

```

    end

    PRESETn = 1;
    forever begin
        #10ns PCLK = ~PCLK;
    end
end

endmodule: top_tb

```

## Top-Level HDL Domain

```

module top_hdl;

// PCLK and PRESETn
//
logic PCLK;
logic PRESETn;

//
// Instantiate the interfaces:
//
apb_if APB(PCLK, PRESETn); // APB interface
spi_if SPI(PCLK, PRESETN); // SPI Interface
intr_if INTR(PCLK, PRESETn); // Interrupt

apb_driver_bfm APB_DRIVER(APB.driver_mp);
apb_monitor_bfm APB_MONITOR(APB.monitor_mp);

spi_driver_bfm SPI_DRIVER(SPI.driver_mp);
spi_monitor_bfm SPI_MONITOR(SPI.monitor_mp);

//
// Add pertinent virtual interfaces to the UVM configuration database:
//
// tbx vif_binding_block
initial begin
    import uvm_pkg::uvm_config_db;

    uvm_config_db #(virtual intr_if)::set(null, "uvm_test_top", $psprintf("%m.INTR"), INTR);

    uvm_config_db #(virtual apb_driver_bfm)::set(null, "uvm_test_top", $psprintf("%m.APB_DRIVER"), APB_DRIVER);
    uvm_config_db #(virtual apb_monitor_bfm)::set(null, "uvm_test_top", $psprintf("%m.APB_MONITOR"), APB_MONITOR);

    uvm_config_db #(virtual spi_driver_bfm)::set(null, "uvm_test_top", $psprintf("%m.SPI_DRIVER"), SPI_DRIVER);
    uvm_config_db #(virtual spi_monitor_bfm)::set(null, "uvm_test_top", $psprintf("%m.SPI_MONITOR"), SPI_MONITOR);

```

```
end

// DUT
spi_top DUT(
  // Port connections not shown...
);

//


// Clock and reset initial blocks:
//


// tbx clkgen
initial begin
  PCLK = 0;
  forever begin
    #10ns PCLK = ~PCLK;
  end
end

// tbx clkgen
initial begin
  PRESETn = 0;
  #80 PRESETn = 1;
end

endmodule: top_hdl
```

## Top-level Testbench Domain

```
module top_tb;

import uvm_pkg::*;
import spi_test_lib_pkg::*;

// UVM initial block: run_test()
initial begin
  run_test();
end

endmodule: top_tb
```

# Debug of SV and UVM

## BuiltInDebug

While simulators should provide (and do provide in the case of QuestaSim<sup>[1]</sup>) useful debug capabilities to help diagnose issues when they happen in a UVM testbench, it also is useful to know the built-in debug features that come with the UVM library. UVM provides a vast code base with ample opportunities for subtle issues. This article will explore the functions and plusargs that are available out of the box with UVM to help with debug.

### Configuration Debug Features

The UVM Resource Database is used to pass configuration information from a test down into a testbench. It is one of the ways that the test controls "what" is going to happen in the testbench leaving the testbench to decide "how" it is going to happen. This mechanism is very powerful, but it does rely on string matching to function correctly. To that end, UVM provides a few capabilities to help ensure those strings match up.

### UVM Command Line Processor

The UVM Command Line Processor can be used to turn on trace messages which then print out UVM\_INFO messages showing when information is placed into the resource database (a set) or pulled out of the resource database (a get). The command line processor provides two plusargs: +UVM\_RESOURCE\_DB\_TRACE and +UVM\_CONFIG\_DB\_TRACE. +UVM\_RESOURCE\_DB\_TRACE is used when the uvm\_resource\_db API is used in the SystemVerilog source code. +UVM\_CONFIG\_DB\_TRACE is used when the uvm\_config\_db API is used. When turned on, the output will look something like this:

```
UVM_INFO ../../uvm-1.1a/src/base/uvm_resource_db.svh(129) @ 0 ns: reporter [CFGDB/SET]
Configuration 'uvm_test_top.mem_interface' (type virtual mem_interface) set by =
(virtual mem_interface) ?
UVM_INFO ../../uvm-1.1a/src/base/uvm_resource_db.svh(129) @ 0 ns: reporter [CFGDB/GET]
Configuration 'uvm_test_top.mem_interface' (type virtual mem_interface) read by
uvm_test_top = (virtual mem_interface) ?
```

### UVM Component

Additionally each class derived from uvm\_component which would include drivers, monitors, agents, environments, etc. comes with some built-in functions to help with configuration debug.

Function	Prototype	Description
print_config	function void print_config( bit recurse = 0, bit audit = 0 )	Print_config_settings prints all configuration information for this component, as set by previous calls to set_config_* and exports to the resources pool. The settings are printing in the order of their precedence. If <i>recurse</i> is set, then configuration information for all children and below are printed as well. If <i>audit</i> is set then the audit trail for each resource is printed along with the resource name and value
print_config_with_audit	function void print_config_with_audit( bit recurse = 0 )	Operates the same as print_config except that the audit bit is forced to 1. This interface makes user code a bit more readable as it avoids multiple arbitrary bit settings in the argument list. If <i>recurse</i> is set, then configuration information for all children and below are printed as well.

These functions could be called from the build\_phase, connect\_phase or most likely the end\_of\_elaboration\_phase. These functions can also be used without modifying the source code by using the Questa specific "call" TCL command. The "call" command allows for SV functions to be called from the TCL command line. An example of using the call command to print the config would look like this:

```
call [examine -handle
{sim:/uvm_pkg::uvm_top.top_levels[0].super.m_env.m_mem_agent.m_monitor}].print_config
```

## Resource Database Dump

Another option for exploring what is currently in the resource database is the dump() command. When used, it will print out what is currently in the resource database along with the types and paths that are stored for each item. To use this facility, uvm\_config\_db #(〈type〉)::dump() needs to be added to your source code. "〈type〉" can be anything including int, bit, or some user defined type. When this is added, output similar to the following will be seen.

```
# === resource pool ===
# mem_config [/^uvm_test_top\..*mem_agent.*$/] : (class mem_agent_pkg::mem_config)
#
# -----
# Name      Type      Size  Value
# -----
# m_mem_cfg  mem_config  -      @552
# -----
# -
# -
# mem_interface [/^uvm_test_top$/] : (virtual mem_interface) ?
# -
# mem_intf_mon_mp [/^uvm_test_top$/] : (virtual mem_interface.monitor_mp) ?
```

```
# -
# === end of resource pool ===
```

This function could be called from the build\_phase, connect\_phase or most likely the end\_of\_elaboration\_phase.

## Factory Debug Features

The UVM Factory is used to create objects in UVM. It also allows the test another mechanism for controlling "what" is going to happen in a testbench by the use of factory overrides. The Factory is created automatically as a singleton object with a handle called "factory" living in the uvm\_pkg. This means that functions can be called on this factory singleton to help understand who is registered with the Factory, which overrides the Factory currently is using and to test out what objects would be returned by the factory for a given type. There are three functions which provide this information.

Function	Prototype	Description
print	function void print ( int all_types = 1 )	Prints the state of the uvm_factory, including registered types, instance overrides, and type overrides. When <i>all_types</i> is 0, only type and instance overrides are displayed. When <i>all_types</i> is 1 (default), all registered user-defined types are printed as well, provided they have names associated with them. When <i>all_types</i> is 2, the UVM types (prefixed with uvm_) are included in the list of registered types.

debug_create_by_type	<pre>function void debug_create_by_type ( uvm_object_wrapper requested_type,                                     string parent_inst_path = "",                                     string name = "" )</pre>	<p>These methods perform the same search algorithm as the <code>create_*</code> methods, but they do not create new objects. Instead, they provide detailed information about what type of object it would return, listing each override that was applied to arrive at the result. When requesting by type, the <code>requested_type</code> is a handle to the type's proxy object. Preregistration is not required.</p>
debug_create_by_name	<pre>function void debug_create_by_name ( string requested_type_name,                                     string parent_inst_path = "",                                     string name = "" )</pre>	<p>When requesting by name, the <code>request_type_name</code> is a string representing the requested type, which must have been registered with the factory with that name prior to the request. If the factory does not recognize the <code>requested_type_name</code>, an error is produced and a null handle returned.</p> <p>If the optional <code>parent_inst_path</code> is provided, then the concatenation, <code>{parent_inst_path, ".","~name~"}</code>, forms an instance path (context) that is used to search for an instance override. The <code>parent_inst_path</code> is typically obtained by calling the <code>uvm_component::get_full_name</code> on the parent.</p>

The most commonly used of these function is `print`. That is used in the form `factory.print()` which returns information similiar to the following which includes which classes are registered with the factory and any overrides which are registered with the factory.

```
##### Factory Configuration (*)
#
# No instance overrides are registered with this factory
#
# Type Overrides:
#
#   Requested Type   Override Type
#   -----  -----
#   mem_seq_base      mem_seq1
#
```

```

# All types registered with the factory: 72 total
# (types without type names will not be printed)
#
#      Type Name
# -----
# analysis_group
# coverage
# directed_test1
# environment
# mem_agent
# mem_config
# mem_driver
# mem_item
# mem_monitor
# mem_seq1
# mem_seq2
# mem_seq_base
# mem_seq_lib
# mem_trans_recorder
# questa_uvm_recorder
# scoreboard
# test1
# test_base
# test_predictor
# test_seq_lib
# threaded_scoreboard
# (*) Types with no associated type name will be printed as <unknown>
#
#####

```

These functions could be called from the build\_phase, connect\_phase or most likely the end\_of\_elaboration\_phase. Factory.print() could also be called using the Questa "call" command like this:

```
call /uvm_pkg::factory.print
```

## Phasing Debug Features

Phasing in UVM can be complicated with multiple phases running in parallel. To help understand when a phase starts and ends, the UVM Command Line Processor can be used to enable a trace with the +UVM\_PHASE\_TRACE plusarg.

When this plusarg is added to the simulator command line, it results in output similiar to this:

```

# UVM_INFO ../../uvm-1.1a/src/base/uvm_phase.svh(1364) @ 0 ns: reporter [PH/TRC/SCHEDULED]
Phase 'common.run' (id=93) Scheduled from phase common.start_of_simulation
# UVM_INFO ../../uvm-1.1a/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'common.run' (id=93) Starting phase
# UVM_INFO ../../uvm-1.1a/src/base/uvm_phase.svh(1364) @ 0 ns: reporter [PH/TRC/SCHEDULED]

```

```

Phase 'uvm' (id=142) Scheduled from phase common.start_of_simulation
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'uvm' (id=142) Starting phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1342) @ 0 ns: reporter [PH/TRC/DONE] Phase
'uvm' (id=142) Completed phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1364) @ 0 ns: reporter [PH/TRC/SCHEDULED]
Phase 'uvm.uvm_sched' (id=154) Scheduled from phase uvm
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched' (id=154) Starting phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1342) @ 0 ns: reporter [PH/TRC/DONE] Phase
'uvm.uvm_sched' (id=154) Completed phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1364) @ 0 ns: reporter [PH/TRC/SCHEDULED]
Phase 'uvm.uvm_sched.pre_reset' (id=172) Scheduled from phase uvm.uvm_sched
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched.pre_reset' (id=172) Starting phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1191) @ 0 ns: reporter [PH/TRC/SKIP] Phase
'uvm.uvm_sched.pre_reset' (id=172) No objections raised, skipping phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1191) @ 0 ns: reporter [PH/TRC/SKIP] Phase
'common.run' (id=93) No objections raised, skipping phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1342) @ 0 ns: reporter [PH/TRC/DONE] Phase
'uvm.uvm_sched.pre_reset' (id=172) Completed phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1364) @ 0 ns: reporter [PH/TRC/SCHEDULED]
Phase 'uvm.uvm_sched.reset' (id=184) Scheduled from phase uvm.uvm_sched.pre_reset
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched.reset' (id=184) Starting phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1342) @ 80 ns: reporter [PH/TRC/DONE] Phase
'uvm.uvm_sched.reset' (id=184) Completed phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1364) @ 80 ns: reporter [PH/TRC/SCHEDULED]
Phase 'uvm.uvm_sched.post_reset' (id=196) Scheduled from phase uvm.uvm_sched.reset
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1114) @ 80 ns: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched.post_reset' (id=196) Starting phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1191) @ 80 ns: reporter [PH/TRC/SKIP] Phase
'uvm.uvm_sched.post_reset' (id=196) No objections raised, skipping phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1342) @ 80 ns: reporter [PH/TRC/DONE] Phase
'uvm.uvm_sched.post_reset' (id=196) Completed phase

```

## Objection Debug Features

Objections are used to control when a time consuming phase is going to end. Understanding when an objection is raised or lowered can be a difficult proposition especially since all of the raise objection calls must be matched with an equal number of drop objection calls. The UVM Command Line Processor can be used to enable a trace with the `+UVM_OBJECTION_TRACE` plusarg. When this plusarg is added to the simulator command line, it results in output similiar to this:

```
# UVM_INFO @ 0 ns: reset_objection [OBJTN_TRC] Object uvm_test_top raised 1 objection(s)
(Raising Reset Objection): count=1 total=1
# UVM_INFO @ 0 ns: reset_objection [OBJTN_TRC] Object uvm_top added 1 objection(s) to its
total (raised from source object uvm_test_top, Raising Reset Objection): count=0
total=1
# UVM_INFO @ 80 ns: reset_objection [OBJTN_TRC] Object uvm_test_top dropped 1
objection(s) (Dropping Reset Objection): count=0 total=0
# UVM_INFO @ 80 ns: reset_objection [OBJTN_TRC] Object uvm_test_top all_dropped 1
objection(s) (Dropping Reset Objection): count=0 total=0
# UVM_INFO @ 80 ns: reset_objection [OBJTN_TRC] Object uvm_top subtracted 1 objection(s)
from its total (dropped from source object uvm_test_top, Dropping Reset Objection):
count=0 total=0
# UVM_INFO @ 80 ns: reset_objection [OBJTN_TRC] Object uvm_top subtracted 1 objection(s)
from its total (all_dropped from source object uvm_test_top, Dropping Reset Objection):
count=0 total=0
# UVM_INFO @ 80 ns: main_objection [OBJTN_TRC] Object uvm_test_top raised 1 objection(s)
(Raising Main Objection): count=1 total=1
# UVM_INFO @ 80 ns: main_objection [OBJTN_TRC] Object uvm_top added 1 objection(s) to its
total (raised from source object uvm_test_top, Raising Main Objection): count=0 total=1
# UVM_INFO @ 5350 ns: main_objection [OBJTN_TRC] Object uvm_test_top dropped 1
objection(s) (Dropping Main Objection): count=0 total=0
# UVM_INFO @ 5350 ns: main_objection [OBJTN_TRC] Object uvm_test_top all_dropped 1
objection(s) (Dropping Main Objection): count=0 total=0
# UVM_INFO @ 5350 ns: main_objection [OBJTN_TRC] Object uvm_top subtracted 1 objection(s)
from its total (dropped from source object uvm_test_top, Dropping Main Objection):
count=0 total=0
# UVM_INFO @ 5350 ns: main_objection [OBJTN_TRC] Object uvm_top subtracted 1 objection(s)
from its total (all_dropped from source object uvm_test_top, Dropping Main Objection):
count=0 total=0
```

## TLM Port Debug Features

Components in UVM are connected together via TLM ports, exports and imps. UVM provides two functions which can be called on a port, export or imp to help understand which objects are connected together. Those two functions are `debug_connected_to()` (usually used with ports) and `debug_provided_to()` (usually used with imps or exports).

Function	Prototype	Description
debug_connected_to	function void debug_connected_to ( int level = 0, int max_level = -1 )	The debug_connected_to method outputs a visual text display of the port/export/imp network to which this port connects (i.e., the port's fanout). This method must not be called before the end_of_elaboration phase, as port connections are not resolved until then.
debug_provided_to	function void debug_provided_to ( int level = 0, int max_level = -1 )	The debug_provided_to method outputs a visual display of the port/export network that ultimately connect to this port (i.e., the port's fanin). This method must not be called before the end_of_elaboration phase, as port connections are not resolved until then.

These functions should be called from the end\_of\_elaboration\_phase as that is when all connections will have been completed by then.

The debug\_connected\_to() function could also be used in a call command like this:

```
call [examine -handle
{sim:/uvm_pkg::uvm_top.top_levels[0].super.m_env.m_mem_agent.m_monitor.result_from_monitor_ap}].debug_connected_to
```

When debug\_connected\_to() is used on a port, output similar to the following should be seen:

```
# UVM_INFO @ 0 ns: uvm_test_top.m_env.m_mem_agent.m_monitor.result_from_monitor_ap
[debug_connected_to] This port's fanout network:
#
#   uvm_test_top.m_env.m_mem_agent.m_monitor.result_from_monitor_ap (uvm_analysis_port)
#   |
#   |_uvm_test_top.m_env.m_mem_agent.m_mon_out_ap (uvm_analysis_port)
#   |
#   |_uvm_test_top.m_env.m_analysis.analysis_export (uvm_analysis_export)
#   |
#   |_uvm_test_top.m_env.m_analysis.coverage_h.analysis_imp (uvm_analysis_imp)
#   |
#   |_uvm_test_top.m_env.m_analysis.test_predictor_h.analysis_imp
(uvm_analysis_imp)

#
#   Resolved implementation list:
#   0: uvm_test_top.m_env.m_analysis.coverage_h.analysis_imp (uvm_analysis_imp)
#   1: uvm_test_top.m_env.m_analysis.test_predictor_h.analysis_imp (uvm_analysis_imp)
```

The debug\_provided\_to() function could also be used in a call command like this:

```
call [examine -handle
{sim:/uvm_pkg::uvm_top.top_levels[0].super.m_env.m_analysis.coverage_h.super.analysis_export}].debug_provided_to
```

When debug\_provided\_to() is used on an export or an imp, output similar to the following should be seen:

```
# UVM_INFO @ 0 ns: uvm_test_top.m_env.m_analysis.coverage_h.analysis_imp
[debug_provided_to] This port's fanin network:
#
#   uvm_test_top.m_env.m_analysis.coverage_h.analysis_imp (uvm_analysis_imp)
#   |
#   |_uvm_test_top.m_env.m_analysis.analysis_export (uvm_analysis_export)
#   |
#   |_uvm_test_top.m_env.m_mem_agent.m_mon_out_ap (uvm_analysis_port)
#   |
#   |_uvm_test_top.m_env.m_mem_agent.m_monitor.result_from_monitor_ap
(uvm_analysis_port)
```

## Callback Debug Features

Callbacks allow for functions and tasks to be called on an external object from a standard object.

### Display Function

If callbacks are used in a testbench, UVM can print out all of the currently registered callbacks using the display function.

Function	Prototype	Description
display	static function void display( T obj = null )	This function displays callback information for obj. If obj is null, then it displays callback information for all objects of type T, including typewide callbacks.

This function should be called from the end\_of\_elaboration\_phase. As an example, to display all the callbacks registered with the uvm\_reg class, the SystemVerilog code would have the following line added:

```
uvm_reg_cb::display();
```

This results in output that looks like the following:

```
# Registered callbacks for all instances of uvm_reg
#
# -----
# status_reg_h_cb    on dut_rm.status_reg_h  ON
# RegA_h_cb          on dut_rm.RegA_h        ON
# RegB_h_cb          on dut_rm.RegB_h        ON
```

## Compile Time Option

Additionally, if the uvm\_pkg is being compiled manually, another option exists which will enable tracing of callbacks. When compiling the uvm\_pkg, +define+UVM\_CB\_TRACE\_ON can be added which turns on output similar to the following:

```
UVM_INFO ..../uvm-1.1a/src/base/uvm_callback.svh(1142) @ 120 ns: reporter [UVMCB_TRC]
Callback mode for status_reg_h_cb is ENABLED : callback status_reg_h_cb
(uvm_callback@837)
```

This is displayed when a callback is accessed in simulation.

## General Debug Features

Some other functions and techniques are available as well.

### Component Hierarchy

To print out the current component hierarchy of a testbench, the print\_topology() function can be called.

Function	Prototype	Description
print_topology	function void print_topology ( uvm_printer printer = null )	Print the verification environment's component topology. The printer is a uvm_printer object that controls the format of the topology printout; a null printer prints with the default output.

Generally, this function would be called from the end\_of\_elaboration phase. There is also a bit called enable\_print\_topology which defaults to 0 and is a data member of the uvm\_root class. When this bit is set to a 1, the entire testbench topology is printed just after completion of the end\_of\_elaboration phase. Since the uvm\_pkg::uvm\_top handle points to a singleton of the uvm\_root class, this bit could be set by adding

```
uvm_top.enable_print_topology = 1;
```

to the testbench code. The print\_topology function can also be called from the Questa command line using the "call" command like this:

```
call /uvm_pkg::uvm_top.print_topology
```

### Verbosity Controls

The messaging system in UVM also provides a robust way of controlling verbosity. Useful UVM\_INFO messages could be sprinkled throughout the testbench with their verbosity set to either UVM\_HIGH or UVM\_DEBUG which would cause them to normally not be printed. When a user would like to have these messages be displayed, then the UVM Command Line Processor comes into play. It has a global verbosity setting which is available using the +UVM\_VERBOSITY plusarg or individual components can have their verbosity settings altered by using the +uvm\_set\_verbosity plusarg.

## DVCon Papers

Papers have also been written about how to debug class based environments. One paper which was the runner up for the DVCon 2012 best paper award is Better Living Through Better Class-Based SystemVerilog Debug [2]. This paper contains several other useful techniques which can be employed to help with UVM debug.

# Reporting/Verbosity

---

## Introduction

UVM provides a built-in mechanism to control how many messages are printed in a UVM based testbench. This mechanism is based on comparing integer values specified when creating a debug message using either the `uvm_report_info()` function or the ``uvm_info()` macro.

## Verbosity Levels

UVM defines an enumerated value (`uvm_verbosity`) which provides several predefined levels. These levels are used in two places. The first place is when writing the message in the code. The second place is the setting that every `uvm_component` possesses to determine a message should be displayed or filtered.

Message Verbosity Setting	Component Verbosity Setting					
	UVM_NONE	UVM_LOW	UVM_MEDIUM	UVM_HIGH	UVM_FULL	UVM_DEBUG
UVM_NONE	Displayed	Displayed	Displayed	Displayed	Displayed	Displayed
UVM_LOW	Filtered	Displayed	Displayed	Displayed	Displayed	Displayed
UVM_MEDIUM	Filtered	Filtered	Displayed	Displayed	Displayed	Displayed
UVM_HIGH	Filtered	Filtered	Filtered	Displayed	Displayed	Displayed
UVM_FULL	Filtered	Filtered	Filtered	Filtered	Displayed	Displayed
UVM_DEBUG	Filtered	Filtered	Filtered	Filtered	Filtered	Displayed

The default verbosity level out of the box is `UVM_MEDIUM` which means that every message with verbosity level of `UVM_MEDIUM`, `UVM_LOW` or `UVM_NONE` will be displayed. As you set a components verbosity level to a higher setting, it will expose more detail (become more verbose) in the transcript. This is a bit counter-intuitive for a lot of people as they expect messages with a "higher" verbosity setting to always be printed. In UVM, messages with a "lower" verbosity setting will be printed before messages with a higher setting. In fact, a message with a `UVM_NONE` setting will always be printed.

Components contain a verbosity setting which is compared against to determine if a message will be displayed or not. There is also the `uvm_top` component whose verbosity setting is used when messages are composed in other objects such as configuration objects which are not derived from `uvm_component` or if a message is composed in a module.

Sequences have their own set of rules for determining which component to use for accessing the current verbosity setting. If a sequence is running on a sequencer (`sequence.start(sequencer)`), then the sequence uses the verbosity setting of the sequencer it is running on. If a sequence is started with a null sequencer handle (for example a virtual sequence), then `uvm_top`'s verbosity setting is used. Sequence Items follow the similar rules. When a sequence item is created in a sequence, it will use the verbosity setting of the sequencer the sequence is running on if the sequencer handle is not null. Otherwise, the sequence item defaults to using `uvm_top`'s verbosity setting.

Sequences and sequence items behave differently because `uvm_report_info/warning/error/fatal()` functions are defined in the `uvm_sequence_item` class which `uvm_sequence` inherits from. Inside these functions, there is a check to see if the sequencer handle is null or not. If the sequencer handle is not null, it's verbosity setting is used. If it is null, then `uvm_top`'s verbosity setting is used. This check is performed the first time a `uvm_report_info/warning/error/fatal()` function is called.

**Note:** If a sequence is started on one sequencer, prints out a message, finishes and then is started on another sequencer, the original sequencer's verbosity setting will be used. This is an issue in UVM which will be fixed in a future release.

## Usage

To take advantage of the different verbosity levels, a testbench needs to use the `uvm\_info macro (which calls the uvm\_report\_info() function). The third argument is what is used for controlling verbosity. For example:

```
task run_phase(uvm_phase phase);
  `uvm_info({get_type_name(),":run_phase"}, "Starting run_phase", UVM_HIGH)
  ...
  `uvm_info({get_type_name(),":run_phase"}, "Checkpoint 1 of run_phase", UVM_MEDIUM)
  ...
  `uvm_info({get_type_name(),":run_phase"}, "Checkpoint 2 of run_phase", UVM_LOW)
  ...
  `uvm_info({get_type_name(),":run_phase"}, "Ending run_phase", UVM_HIGH)
endtask : run_phase
```

Running this code with default settings would only result in the Checkpoint 1 and Checkpoint 2 messages being printed.

**Note:** only "info" messages can be masked using verbosity settings. "warning", "error" and "fatal" messages will always be printed.

An additional way to use verbosity settings is to explicitly check what the current verbosity setting is. The uvm\_report\_enabled() function will return a 0 or a 1 to inform the user if the component is currently configured to print messages at the verbosity level passed in. Code can then explicitly check the current verbosity setting before calling functions which display information (item.print(), etc.) which don't take into account verbosity settings.

```
task run_phase(uvm_phase phase);
  ...
  if (uvm_report_enabled(UVM_HIGH))
    item.print();
  ...
endtask : run_phase
```

Here we are checking if UVM\_HIGH messages should be printed. If they should be printed, then we will print out the item.

## Message ID Guidance

Every message that is printed contains an ID which is associated with the message. The ID field can be used for any kind of user-specified identification or filtering, outside the simulation. Within the simulation, the ID can be used alongside the severity, verbosity attributes to configure any report's actions, output file(s), or make other decisions to modify or suppress the report. This capability is enhanced further in UVM with the UvmMessageCatching API.

Since verbosity can get down to an ID level of granularity, it is recommended to follow the pattern of concatenating `get_type_name()` together with a secondary string. The static function `get_type_name()` (created by ``uvm_component/object_utils()` macro) will inform the user which class the string is coming from. The secondary string can then be used to identify the function/task where the message was composed or for further granularity as needed.

## Verbosity Controls

With messages in place which will use verbosity settings, the functions and plusargs which allow controlling the current verbosity setting need to be discussed. These functions and plusargs can be divided into global controls and more fine grain controls.

### Global Controls

To globally set a verbosity level for a testbench from the test on down, the `set_report_verbosity_level_hier()` function would be used.

```
// Function: set_report_verbosity_level_hier
//
// This method recursively sets the maximum verbosity level for reports for
// this component and all those below it. Any report from this component
// subtree whose verbosity exceeds this maximum will be ignored.
//
// See <uvm_report_handler> for a list of predefined message verbosity levels
// and their meaning.

extern function void set_report_verbosity_level_hier (int verbosity);
```

This function when called from a testbench module will globally set the verbosity setting for the entire UVM testbench. This would look like this:

```
module testbench;
  import uvm_pkg::*;
  import uvm_tests_pkg::*;

  //Dut instantiation, etc.

  initial begin
    // Other TB initialization code such as config_db calls to pass virtual
    // interface handles to the testbench

    //Turn on extra debug messages
```

```

uvm_top.set_report_verbosity_level(UVM_HIGH);

//Run the test (UVM_TESTNAME will override "test1" if set on cmd line)
run_test("test1");
end

endmodule : testbench

```

An even easier way which doesn't require recompilation is to use the UVM Command Line Processor to process the +UVM\_VERBOSITY plusarg. This would look like this:

```
vsim testbench +UVM_VERBOSITY=UVM_HIGH
```

## Fine Grain Controls

In addition to the global controls, UVM allows for individual setting of verbosity levels at a component level and even at an ID level of granularity.

To set an individual components verbosity level, the `set_report_verbosity_level()` function can be used.

```

// Function: set_report_verbosity_level
//
// This method sets the maximum verbosity level for reports for this component.
// Any report from this component whose verbosity exceeds this maximum will
// be ignored.

function void set_report_verbosity_level (int verbosity_level);

```

There is also a hierarchical version of this function which will set a component's and its children's verbosity level.

```
function void set_report_verbosity_level_hier ( int verbosity );
```

The UVM Command Line Processor also provides for more fine grain control by using the +uvm\_set\_verbosity plusarg. This plusarg allows for component level and/or ID level verbosity control.

There are several other functions which can be used to change a specific ID's verbosity level, change actions for specific severities, etc. These methods are documented in the UVM Reference guide.

## Performance Considerations

This article mentions both the `uvm_report_info()` function and the ``uvm_info()` macro. Mentor Graphics recommends using the ``uvm_info()` macro for dealing with reporting for a couple reasons (detailed in the Macro Cost Benefit DVCon paper). The most important of those reasons is that the ``uvm_info()` macro can significantly speed up simulations when a verbosity setting is such that messages won't be printed. The reason for the speed up is the macro checks the verbosity setting before doing any string processing.

Looking at a simple ``uvm_info()` statement:

```
`uvm_info("Message_ID", $sformatf("%s, data[%0d] = 0x%08x", name, ii, data[ii]), UVM_HIGH)
```

there is a fairly complex `$sformat` statement contained within the ``uvm_info()` statement. The `$sformat()` statement would take enough simulation time to be noticeable when called repeatedly. With a verbosity setting of `UVM_HIGH`, this

`$format()` message won't be printed most of the time. Because of the macro usage, the processing time required to generate the resultant string from processing the `$format()` message won't be wasted as well.

The ``uvm_info()` macro under the hood is using the `uvm_report_object::uvm_report_enabled()` API to perform an inexpensive check of the verbosity setting to determine if a message will be printed or not before ultimately calling `uvm_report_info()` with the same arguments as the macro if the check returns a positive result.

```
if (uvm_report_enabled(UVM_HIGH, UVM_INFO, "Message_ID"))
    uvm_report_info("Message_ID", $formatf("%s, data[%0d] = 0x%08x", name, ii, data[ii]), UVM_HIGH);
```

If `uvm_report_info()` was used directly instead

```
uvm_report_info("Message_ID", $formatf("%s, data[%0d] = 0x%08x", name, ii, data[ii]), UVM_HIGH);
```

the `$format()` string would have been processed before entering the `uvm_report_info()` function even though ultimately that string will not be printed due to the `UVM_HIGH` verbosity setting.

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# UVM/CommandLineProcessor

---

## Introduction

UVM adds several new features compared to OVM. One of the new features is a command line processor which is used to easily interact with plusargs. Several plusargs are pre-defined and part of the UVM standard. The predefined plusargs allow for modification of built-in UVM variables including verbosity settings, setting integers and strings in the resource database, and controlling tracing for phases and resource database accesses. The command line processor also eases interacting with user defined plusargs.

In general, plusargs should be used only to temporarily change what is going to happen in a testbench. For example, to enable additional debug information when something goes wrong or to control transaction recording when running in GUI mode.

## Built-In UVM Plusargs

There are two classes of built-in UVM plusargs. The first class are plusargs which can only be set once on the command line. The second class are plusargs which can be set multiple times.

### Single Use Plusargs

Plusarg Name	Description	Example Usage
+UVM_TESTNAME	+UVM_TESTNAME=<class name> allows the user to specify which uvm_test (or uvm_component) should be created via the factory and cycled through the UVM phases. If multiple of these settings are provided, the first occurrence is used and a warning is issued for subsequent settings.	vsim testbench +UVM_TESTNAME="block_test1"
+UVM_VERBOSITY	+UVM_VERBOSITY=<verbosity> allows the user to specify the initial verbosity for all components. The uvm_verbosity enum values (UVM_LOW, UVM_MEDIUM, UVM_HIGH, etc.) and integer values are accepted as arguments. If multiple of these settings are provided, the first occurrence is used and a warning is issued for subsequent settings.	vsim testbench +UVM_VERBOSITY=UVM_HIGH
+UVM_TIMEOUT	+UVM_TIMEOUT=<timeout>,<overridable> allows users to change the global timeout of the UVM framework. The timeout value is specified as an integer number of ns. Time specifiers such as ms or us can not be used currently. The <overridable> argument ('YES' or 'NO') specifies whether user code can subsequently change this value. If set to 'NO' and the user code tries to change the global timeout value, a warning message will be generated. The default value for <overridable> is 'YES'.	vsim testbench +UVM_TIMEOUT=1000000,NO

+UVM_MAX_QUIT_COUNT	+UVM_MAX_QUIT_COUNT=<count>,<overridable> allows users to change max quit count for the report server. The <overridable> argument ('YES' or 'NO') specifies whether user code can subsequently change this value. If set to 'NO' and the user code tries to change the max quit count value, an warning message will be generated. The default value for <overridable> is 'YES'.	vsim testbench +UVM_MAX_QUIT_COUNT=5, NO
+UVM_PHASE_TRACE	+UVM_PHASE_TRACE turns on tracing of phase executions. Users simply need to put the argument on the command line.	vsim testbench +UVM_PHASE_TRACE
+UVM_OBJECTION_TRACE	+UVM_OBJECTION_TRACE turns on tracing of objection activity. If a description was supplied when raising or dropping an objection, it will be displayed when this plusarg is set. Users simply need to put the argument on the command line.	vsim testbench +UVM_OBJECTION_TRACE
+UVM_RESOURCE_DB_TRACE	+UVM_RESOURCE_DB_TRACE turns on tracing of resource DB access. Users simply need to put the argument on the command line.	vsim testbench +UVM_RESOURCE_DB_TRACE
+UVM_CONFIG_DB_TRACE	+UVM_CONFIG_DB_TRACE turns on tracing of configuration DB access. Users simply need to put the argument on the command line.	vsim testbench +UVM_CONFIG_DB_TRACE

## Multiple Use Plusargs

Plusarg Name	Description	Example Usage
+uvm_set_verbosity	+uvm_set_verbosity=<comp>,<id>,<verbosity>,<phase> and +uvm_set_verbosity=<comp>,<id>,<verbosity>,time,<time> allow the users to manipulate the verbosity of specific components at specific phases (and times during the "run" phases) of the simulation. The id argument can be either _ALL_ for all IDs or a specific message id. Wildcarding is not supported for id due to performance concerns. Settings for non-"run" phases are executed in order of occurrence on the command line. Settings for "run" phases (times) are sorted by time and then executed in order of occurrence for settings of the same time.	vsim testbench \ +uvm_set_verbosity=uvm_test_top.env0.agent1.*,_ALL_,UVM_HIGH,time,0
+uvm_set_action	+uvm_set_action=<comp>,<id>,<severity>,<action> provides the equivalent of various uvm_report_object's set_report_*_action APIs. The special keyword, _ALL_, can be provided for both/either the id and/or severity arguments. The action can be UVM_NO_ACTION or a l separated list of the other UVM message actions (UVM_DISPLAY, UVM_LOG, UVM_COUNT, UVM_EXIT, UVM_CALL_HOOK, UVM_STOP).  <b>Note:</b> As of UVM 1.1, this plusarg can only take a single UVM message action. You can not have a l separated list of UVM message actions due to a bug in the UVM code.	vsim testbench \ +uvm_set_action=uvm_test_top.env0.*,_ALL_,UVM_ERROR,UVM_NO_ACTION

+uvm_set_severity	+uvm_set_severity=<comp>,<id>,<current severity>,<new severity> provides the equivalent of the various uvm_report_object's set_report_*_severity_override APIs. The special keyword, _ALL_, can be provided for both/either the id and/or current severity arguments.	vsim testbench \ +uvm_set_severity=uvm_test_top.env0.*,BAD_CRC,UVM_ERROR,UVM_WARNING
+uvm_set_inst_override +UVM_SET_INST_OVERRIDE +uvm_set_type_override +UVM_SET_TYPE_OVERRIDE	+uvm_set_inst_override=<req_type>,<override_type>,<full_inst_path> and +uvm_set_type_override=<req_type>,<override_type>[,<replace>] work like the name based overrides in the factory--factory.set_inst_override_by_name() and factory.set_type_override_by_name(). For uvm_set_type_override, the third argument is 0 or 1 (the default is 1 if this argument is left off); this argument specifies whether previous type overrides for the type should be replaced.	vsim testbench \ +uvm_set_type_override=eth_packet,short_eth_packet
+uvm_set_config_int +UVM_SET_CONFIG_INT +uvm_set_config_string +UVM_SET_CONFIG_STRING	+uvm_set_config_int=<comp>,<field>,<value> and +uvm_set_config_string=<comp>,<field>,<value> work like their procedural counterparts: set_config_int() and set_config_string(). For the value of int config settings, 'b (0b), 'o, 'd, 'h ('x or 0x) as the first two characters of the value are treated as base specifiers for interpreting the base of the number. Size specifiers are not used since SystemVerilog does not allow size specifiers in string to value conversions.	vsim testbench \ +uvm_set_config_int=*,recording_detail,400

## User Defined Plusargs

In addition to the built-in plusargs that UVM provides, users can take advantage of the command line processor to add new plusargs. These new plusargs could be used in many ways including to pass in filenames for different initial memory images, change the number of iterations in a loop or configure the number of active masters/slaves in a switch or fabric environment.

**Coding Guideline:** Don't prefix user defined plusargs with "uvm\_" or "UVM\_". These prefixes are reserved by the UVM committee for future expansion of the built-in command line argument space. Do consider using a company and/or group prefix to prevent namespace overlap. For example, "MENT\_" could be used by Mentor Graphics employees to denote user defined plusargs created by Mentor Graphics.

To get a value from a plusarg being set once on the command line code like this could be used:

```
uvm_cmdline_processor cmdline_proc = uvm_cmdline_processor::get_inst();  
//get a string  
string my_value = "default_value";  
int rc = cmdline_proc.get_arg_value("+MENT_ABC=", my_value);  
//Convert to an int  
int my_int_value = my_value.atoi();
```

Notice that the value can be converted to an integer by using the standard built-in SV function atoi().

If you want to allow for the possibility of a plusarg being set multiple times, then you can use the get\_arg\_values() function. If the vsim command line looks like this:

```
vsim testbench +MENT_MY_ARG=500,NO +MENT_MY_ARG=200000,YES
```

the testbench could get those values by using the following code snippet.

```
uvm_cmdline_processor cmdline_proc = uvm_cmdline_processor::get_inst();  
//get a string  
string my_values[$];  
int rc = cmdline_proc.get_arg_values("+MENT_MY_ARG=", my_values);
```

This would give me two entries in the my\_values queue. They would be "500,NO" and "200000,YES". I could then write further code to split these strings using the uvm\_split\_string(string str, byte sep, ref string values[\$]). This would look like this:

```
string split_values[$]  
foreach (my_values[i]) begin  
    uvm_split_string(my_values[i], ",", split_values);  
  
    if (split_values[1] == "YES") begin  
        ...  
    end  
end
```

# UVM Connect - SV-SystemC interoperability

## UvmConnect

Guide to UVM Connect Methodology - Learn about how to use UVM Connect to link UVM and SystemC using TLM

### UVM Connect Chapter contents:

[UvmConnect \(this page\)](#) - introduction to UVM Connect

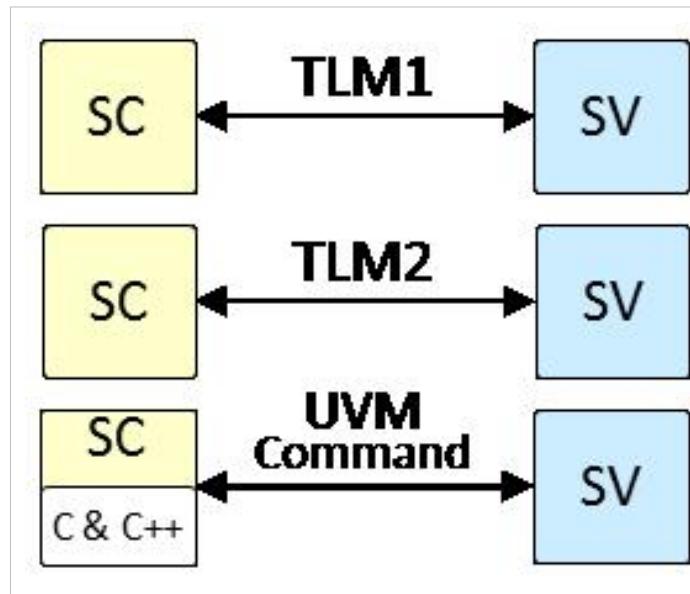
[UvmConnect/Connections](#) - Exchanging data objects in UVM Connect between SystemVerilog and SystemC

[UvmConnect/Conversion](#) - Converting transactions when crossing the SV/SC language boundary

[UvmConnect/CommandAPI](#) - Controlling and/or accessing UVM features from SystemC

### Topic Overview

UVM Connect provides TLM1 and TLM2 connectivity and object passing between SystemC and SystemVerilog models and components. It also provides a UVM Command API for accessing and controlling UVM simulation from SystemC (or C or C++).



## Enabling IP & VIP Reuse

So what does this new capability allow you to do? UVM Connect enables the following use models, all designed to maximize IP reuse:

- Abstraction Refinement - Reuse your SystemC architectural models as reference models in UVM verification. Reuse your stimulus generation agents in SystemVerilog to verify models in SystemC.
- Expansion of VIP Inventory - More off-the-shelf VIP is available when you are no longer confined to VIP written in one language. Increase IP reuse! To properly verify large SoC systems, verification environments are becoming more of an integration problem than a design problem.
- Leveraging language strengths - Each language has its strengths. You can leverage SV's powerful constraint solvers and UVM's sequences to provide random stimulus to your SC architectural models. You can leverage SC's speed and capacity for verification of untimed or loosely timed system-level environments.
- Access to SV UVM from SC - The UVM Command API provides a bridge between SC and UVM simulation in SV. With this API you can wait for and control UVM phase transitions, set and get configuration, issue UVM-style reports, set factory type and instance overrides, and more.

## Key Features

The UVM Connect library makes connecting TLM models in SystemC and UVM in SystemVerilog a relatively straightforward process. This section enumerates some key features of UVM Connect.

- *Simplicity* - Object-based data transfer is accomplished with very little preparation needed by the user.
- *Optional* - The UVMC library is provided as a separate, optional package to UVM. You do not need to import the package if your environments do not require cross-language TLM connections or access to the UVM Command API.
- *Works with Standard UVM* - UVMC works out-of-box with open-source Accellera UVM 1.1a and later. UVMC can also work with previous UVM open-source releases with one minor change.
- *Encourages native modeling methodology* - UVMC does not impose a foreign methodology nor require your models or transactions to inherit from a base class. Your TLM models can fully exploit the features of the language in which they are written.
- *Supports existing models* - Your existing TLM models in both SystemVerilog and SystemC can be reused in a mixed-language context without modification.
- *Reinforces TLM modeling standards* - UVMC reinforces the principles and purpose of the TLM interface standard-designing independent models that communicate without directly referring to each other. Such models become highly reusable. They can be integrated in both native and mixed-language environments without modification.

# UvmConnect/Connections

To communicate, verification components must agree on the data they are exchanging and the interface used to exchange that data. TLM connections parameterized to the type of object (transaction) help components meet these requirements and thus ease integration costs and increase their reuse. To make such connections across the SC-SV language boundary, UVMC provides connect and connect\_hier functions.

SV TLM2:

```
uvmc_tlm #(trans)::connect (port_handle, "lookup");
uvmc_tlm #(trans)::connect_hier (port_handle, "lookup");
```

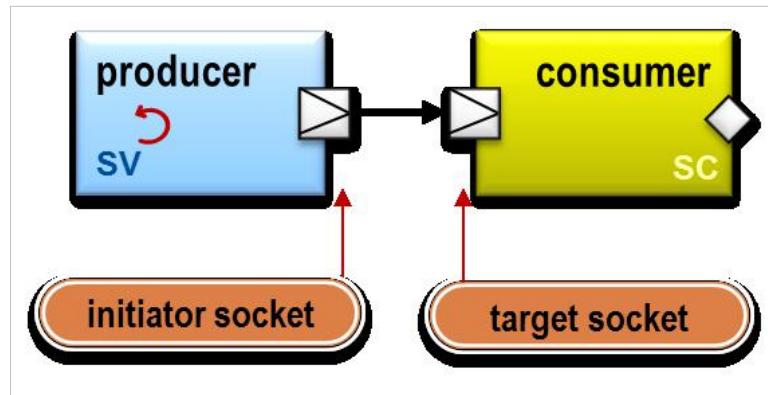
SC TLM2:

```
uvmc_connect (port_ref, "lookup");
uvmc_connect_hier (port_ref, "lookup");
```

- *trans* - Specifies the transaction type for unidirectional TLM 1 ports, export, and imps. Needed only for SV.
- *port\_handle* or *ref* - The handle or reference to the port, export, imp, interface, or socket instance to be connected.
- *lookup* - An optional lookup string to register for this port.

UVMC registers the port's hierarchical name and lookup string (if provided) for later matching against other port registrations within both SV and SC. A string match between any two registered ports results in those ports being connected-whether the components are in the same or different languages.

Let's see how we use the connect function in a real example. The diagram below shows an SV producer component and an SC consumer component communicating via a TLM socket connection.



The following code creates the testbench and UVMC connection in both SV and SC. This code is complete and executable. It is all you need to pass tlm\_generic\_payload transactions between a SystemC and SystemVerilog component.

SystemVerilog:

```
import uvm_pkg::*;
import uvmc_pkg::*;
`include "producer.sv"

module sv_main;
  producer prod = new("prod");
```

```
initial begin
    uvmc_tlm #(())::connect(prod.out, "foo");
    run_test();
end
endmodule
```

SystemC:

```
#include "uvmc.h"
using namespace uvmc;
#include "consumer.h"

int sc_main(int argc, char* argv[]) {
    consumer cons("cons");
    uvmc_connect(cons.in, "foo");
    sc_start(-1);
    return 0;
}
```

The *sv\_main* top-level module creates the SV portion of the example. It creates an instance of a producer component, then registers the producer's out initiator socket with UVMC using the lookup string "foo". It then starts UVM test flow with the call to *run\_test()*.

The *sc\_main* function creates the SC portion of this example. It creates an instance of a consumer *sc\_module*, then registers the consumer's in target socket with UVMC using the lookup string, "foo". It then starts SC simulation with the call to *sc\_start*.

During elaboration, UVMC will connect the producer and consumer sockets because they were registered with the same lookup string.

(Note: SV-side TLM port connections would normally be made in UVM's *connect\_phase*. We omit this for sake of brevity.)

# UvmConnect/Conversion

Object transfer requires converters to translate between the two types when crossing the SC-SV language boundary.

UVMC provides built-in support for the TLM generic payload (TLM GP). You don't need to do anything regarding transaction conversion when using TLM GP. Use of the TLM GP also affords the best opportunity for interoperability between independently designed components from multiple IP suppliers. Thus, there is strong incentive to use the generic payload if possible.

If, however, you are not using the TLM GP, the task of creating a converter in SC and SV is fairly simple. Let's say we have a transaction with a command, address, and variable-length data members. The basic definitions in SV and SC might look like this:

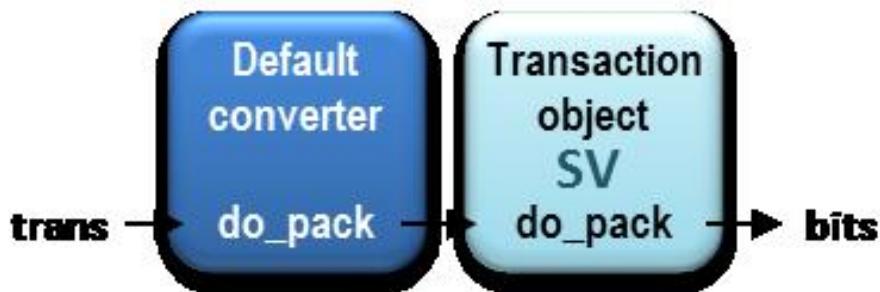
```
SV                               SC
class C;                         class C {
cmd_t cmd;                       cmd_t cmd;
int unsigned addr;                unsigned int addr;
byte data[$];                    vector<char> data;
...
};

endclass
```

UVMC uses separate converter classes to pack and unpack transactions. This allows you to define converters independently from the transactions they operate on. UVMC defines default converter implementations that conform to the prevailing methodology in each language. Yet, UVMC can accommodate almost any application, such as using a SV-side object that does not extend *uvm\_object*.

## SV-side conversion

In SV, by default, UVMC default converter delegates conversion to the transaction's *pack* and *unpack* methods. Having the transaction class itself implement its own conversion is the UVM way, so UVMC accommodates.



UVM transaction conversion is usually implemented in its *do\_pack* and *do\_unpack* methods, or via the *uvm\_field* macros. The *do\_pack/unpack* method implementations have better performance and provide greater flexibility, debug, and type-support than the *uvm\_field* macros. The extra time to implement these methods is far outweighed by the cumulative performance and debug benefits they afford. The recommended way to implement *do\_pack/unpack* is to use a set of `uvm_pack` `uvm_unpack` macros, which are part of the UVM standard. These macros are of the "good" variety. They are small and more efficient than the `uvm_field` macros and `uvm_packer` API. The following transaction definition implements *do\_pack/unpack* in the recommended way:

```

class packet extends uvm_sequence_item;

`uvm_object_utils(packet)

rand cmd_t cmd;
rand int    unsigned addr;
rand byte   data[$];

constraint C_data_size {
    data.size inside {[1:16]};
}

function new(string name="";
super.new(name);
endfunction

virtual function void do_pack(uvm_packer packer);
super.do_pack(packer);
`uvm_pack_int(cmd)
`uvm_pack_int(addr)
`uvm_pack_queue(data)
endfunction

virtual function void do_unpack(uvm_packer packer);
super.do_unpack(packer);
`uvm_unpack_int(cmd)
`uvm_unpack_int(addr)
`uvm_unpack_queue(data)
endfunction

...
endclass

```

## SC-side conversion

Transaction classes in SystemC do not typically extend from a common base class, so conversion is performed in a separate converter class.



An SC-side converter is actually a template specialization of the default converter, *uvmc\_converter<T>*. Template specializations allow you to override the generalized implementation of a parameterized class with one that is custom-tailored for a specific set of parameter values, e.g. *uvmc\_converter<packet>*. This is a compiler-level operation, so any code that uses *uvmc\_converter<packet>* automatically gets our specialized implementation. The following code

implements a converter class for our packet transaction class:

```
#include "uvmc.h"
using namespace uvmc;

template <>
class uvmc_converter<packet> {
public:
    static void do_pack(const packet &t,
                        uvmc_packer &packer) {
        packer << t.cmd << t.addr << t.data;
    }
    static void do_unpack(packet &t,
                          uvmc_packer &packer) {
        packer >> t.cmd >> t.addr >> t.data;
    }
};
```

The packer variable is an instance of *uvmc\_packer*, the counterpart to UVM's *uvm\_packer* on the SV side. You can stream your transaction members are streamed to and from a built-in packer instance variable much like you would stream them to standard out using *cout*. Use *operator<<* for packing, and *operator>>* for unpacking. Except for the actual field names being streamed, most converter classes conform to this template structure. So, as a convenience, UVMC provides macros that can generate a complete *uvmc\_converter<T>* class and output streaming capability for your transaction with a single *UVMC\_UTILS* macro invocation. Unlike the *uvm\_field* macros in UVM, these macros expand into succinct, human-readable code (actually, the exact code shown above). Using the macro option, our custom converter class definition for packet reduces to one line:

```
#include "uvmc.h"
using namespace uvmc;
UVMC_UTILS_3 (packet, cmd, addr, data)
```

The number suffix in the macro name indicates the number of transaction fields being converted. Then, simply list each field member in the order you want them streamed. Just make sure the order is the same as the packing and unpacking that happens on the SV side. UVMC supports up to 20 fields.

## Type Support

For streaming fields in your converter implementations, UVMC supports virtually all the built-in data types in SV and SC. In SV, all the integral types, reals, and single-dimensional, dynamic, queue, and associative arrays of any of the built-in types have direct support. In SC, all the integral types and float/double are support, as are fixed arrays, vectors, lists, and maps of these types. The integral SystemC types such as *sc\_bv<N>* and *sc\_uint<N>* are also supported. Rest assured, any type that does not have direct support can be adapted to one of the supported types.

## On (not) using `uvm\_field macros

The `uvm_field` macros hurt run-time performance, can make debug more difficult, and can not accomodate custom behaviors (e.g. conditional packing, copying, etc. based on value of another field). The macros generate far more code than is necessary to implement the operations directly. This consumes memory and hurts performance. Even a small performance decrease of 1% can dramatically affect regression times and lowers the ceiling on potential accelerator/emulator performance gains.

## Transaction requirements

UVM Connect imposes very few requirements on the transaction types being conveyed between TLM models in SV and SC, a critical requirement for enabling reuse of existing IP. The more restrictions you impose on the transaction type, the more difficult it will be to reuse models that use those transactions.

- No base classes required. It is not required that the transaction inherit from any base class--in either SV or SC--to facilitate their conversion
- No factory registration required. It is not required that the transaction register with a factory--be it the UVM factory or any user-defined factory mechanism.
- No converter API required. It is not required that the transaction provide the conversion methods. You can specify a different converter for each or every UVMC connection. You can define multiple converters for the same transaction type.
- Transaction equivalence not required. It is not required that the members (properties) of the transaction classes in each language be of equal number, equivalent type, and declaration order. The converter can adapt to disparate transaction definitions at the same time it serializes the data.

# UvmConnect/CommandAPI

The UVM Connect Command API gives SystemC users access to key UVM features in SystemVerilog. These include:

- Wait for a UVM to reach a given simulation phase
- Raise and drop objections to phases, effectively controlling UVM test flow
- Set and get UVM configuration, including objects
- Send UVM report messages, and set report verbosity
- Set type and instance overrides in the UVM factory
- Print UVM component topology

To enable use of the UVM Connect command API, you must call *uvmc\_init()* from an *initial* block on the SystemVerilog side. This function starts a background process that services UVM command requests from SystemC.

```
module sv_main;

  import uvm_pkg::*;
  import uvmc_pkg::*;

  initial begin
    uvmc_cmd_init();
    run_test();
  end

endmodule
```

SC-side calls to the UVM Command API will block until SystemVerilog has finished elaboration and the *uvmc\_init()* function has been called. For this reason, such calls must be made from within SC thread processes.

## Issuing UVM reports from SystemC

UVMC provides an API into UVM's reporting mechanism, allowing you to send reports to UVM's report server and to set report verbosity at any context of the UVM hierarchy. As with natively issued UVM reports, all reports you send to UVM are subject to filtration by configured verbosity level, actions, and report catchers.

Just as in UVM, UVMC provides convenient macro definitions for sending reports efficiently and with automatic SystemC-side filename and line number information.

```
UVMC_INFO("SC-TOP/SET_CFG",
  "Setting config for SV-side producer",
  UVM_MEDIUM, "");
```

## Set and Get Config

UVMC supports setting and getting integral, string, and object values from UVM's configuration database.

Use of configuration objects is strongly recommended over one-at-a-time integrals and strings. You can pass configuration for an entire component or set of components with a single call, and the configuration object is type-safe to the components that accept those objects. With ints and strings, it's far too easy to get configuration wrong (which can be

hard to debug), especially with generic field names like "count" and "addr".

Before you can pass configuration objects, you will need to define an equivalent transaction type and converter on the SystemC side. As shown earlier, this is easily done:

```
#include "uvmc.h"
#include "uvmc_macros.h"
using namespace uvmc;

class prod_cfg {
public:
    int min_addr, max_addr;
    int min_data_len, max_data_len;
    int min_trans, max_trans;
};

UVMC_UTILS_6(prod_cfg, min_addr, max_addr,
              min_data_len, max_data_len,
              min_trans, max_trans)
```

We use the above definition to configure a SV-side stimulus generator and a couple of "loose" int and string fields. Note the form is much the same as in UVM.

```
uvmc_set_config_int ("e.agent.driver","",  
                     "max_error", 10);  
  
uvmc_set_config_string ("e.agent.seqr",  
                        "run_phase",  
                        "default_sequence",  
                        s.c_str());  
  
prod_cfg cfg = new();  
cfg.min_addr = 'h0100; cfg.max_addr = 'h0FFF;  
cfg.max_data_len = 10; cfg.max_trans = 100;  
  
uvmc_set_config_object ("prod_cfg", "e.prod",  
                        "", "config", cfg);  
  
if (!uvmc_get_config_int ("sc_top", "dut",  
                         "max_error", max_error))  
    UVMC_ERROR ("NO_MAX_SET",  
               "max_error not set", name());
```

## Phase Control

With UVMC, you can wait for a UVM phase to reach a given state, and you can raise and drop objections to any phase, thus controlling UVM test flow.

```
uvmc_wait_for_phase("run", UVM_PHASE_STARTED);

uvmc_raise_objection("run",
                      "SC producer active");
// produce data

uvmc_drop_objection("run",
                      "SC producer done");
```

## Factory

You can set UVM factory type and instance overrides as well. Let's say you are testing a SC-side slave device and want to drive it with a subtype of the producer that would normally be used. Once you make all your overrides, you can check that they "took" using some factory debug commands.

```
uvmc_set_factory_type_override("producer",
                               "producer_ext", "e.*");
uvmc_set_factory_inst_override("scoreboard",
                               "scoreboard_ext", "e.*");

// factory should show overrides
uvmc_print_factory();

// show information about how the factory
// chooses which type to create. Should be
// the "_ext" versions in this case.

uvmc_debug_factory_create("producer",
                           "e.prod");
uvmc_debug_factory_create("scoreboard",
                           "e.sb");

// get the type the factory would produce
// give a requested type and context. We
// can use the result in subsequent overrides
string override = uvmc_find_factory_override("producer",
                                             "e.prod");
```

## Printing UVM Topology

You can print UVM testbench topology from SystemC. Just be sure you invoke the command after UVM has built the testbench.

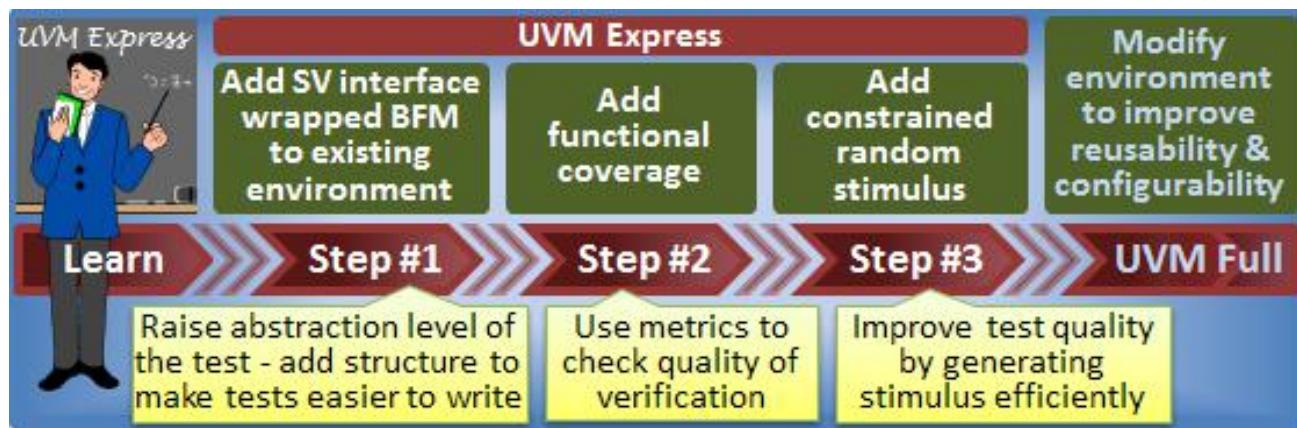
```
uvmc_wait_for_phase("build", UVM_PHASE_ENDED);  
  
cout << "UVM Topology:" << endl;  
  
uvmc_print_topology();
```

# UVM Express - step by step improvement

## UvmExpress

The **UVM Express** Methodology is a collection of techniques, coding styles and UVM usages that are designed to increase the productivity of functional verification. The techniques include raising the abstraction level of tests, writing tests using BFM function and task calls, adding functional coverage, and adding constrained-random stimulus generation.

The UVM Express can be thought of as a series of steps or train stops on journey that may ultimately lead to Full UVM. These collections of steps or train stops are shown below in the diagram.



### UVM Express Chapter contents:

- UvmExpress (this page) - introduction to the UVM Express methodology
- UvmExpress/DUT - the DUT interface
- UvmExpress/BFM - describing the Bus Functional Model
- UvmExpress/WritingBfmTests - how to write first simple tests
- UvmExpress/FunctionalCoverage - adding functional coverage
- UvmExpress/ConstrainedRandom - constrained-random stimulus

## Topic Overview

Each of the train stops will be summarized below, and links provided to more detail. At each point in this train journey value is being added to the testbench. The choice to move to the next stop is an individualized choice. Some users find getting organized with a BFM is good enough. But usually, they continue and add coverage, only to find that coverage is insufficient. Then they move to improving their stimulus and refining their coverage. It is not a requirement that a testbench move along each stop. Nor is it a requirement that a testbench move to "Full UVM".

## Is UVM Express for you?

For those teams/projects involved in designing and verifying digital designs, the questions below might help. If you answer "no" to any of the questions, then UVM Express might be for you. If you answer "yes," then you still might want to begin by using UVM Express, but you'll soon move to Full UVM.

	No	Yes
Do you have a full-time verification lead?	Think about using UVM Express	Consider using full UVM
Do you have separate verification and design teams?	Think about using UVM Express	Consider using full UVM
Are you already using a High-Level Verification Language?	Think about using UVM Express	Consider using full UVM

## UVM Express Introduction

UVM Express will be described in these pages in terms of a worked example. Each step of the process is outlined below. Each topic is explained until finally, our device-under-test has a constrained-random stimulus testbench, with a task-based BFM. The table of contents for UVM Express is listed below in bullet form. Below that, each section of the UVM Express has a short summary paragraph. Additional detail may be found by clicking on the heading for each of the summary sections, or by clicking on the "For More Information" link at the bottom of each summary section.

- Description of the device-under-test
- Description of the BFM
- Writing Tests using BFM Tasks
- Adding Functional Coverage
- Replacing directed tests with Constrained-Random stimulus

The UVM is a class library which provides a view of the SystemVerilog verification features. The UVM class library contains powerful constructs for verification, including a phasing machine, a reporting system, a configuration database, a factory for overriding types and the ability to write tests as sequences (or function calls). Some verification teams are successful adopting the UVM from the beginning, but those teams usually have a background in e, Vera, VMM, AVM or OVM. In order to be proficient with SystemVerilog and the UVM, an investment of time and money needs to be made in training. Often the adoption process can be made shorter by hiring consultants.

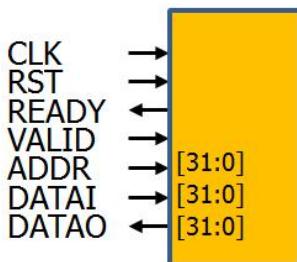
Most verification teams do not have a full-time verification expert on staff, have time and budget restrictions and cannot adopt the UVM in whole or adopt it as quickly as they might like. These teams are usually under-staffed, under-funded and over-worked. They are exactly the kind of people that the UVM is meant to help, but the first step towards adoption is too high.

UVM Express provides a small first step toward UVM adoption. UVM Express is a way to build your testbench environment, a way to raise your abstraction level, a way to check the quality of your tests and a way to think about writing your tests. Each of the steps outlined for UVM Express is a reusable piece of verification infrastructure. These UVM Express steps are a way to progressively adopt a UVM methodology, while getting verification productivity and verification results at each step.

Using UVM Express is not a replacement for full UVM, but instead enables full UVM migration or co-existence at any time. UVM Express is UVM - just organized in a way that allows progressive adoption and a value proposition with each step.

## UVM Express: Writing tests using tasks in a BFM

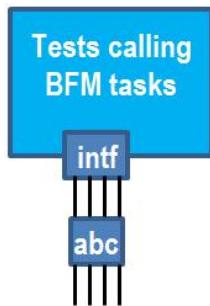
The first level of UVM Express is a BFM (Bus Functional Model). The BFM has a job to manage the signals, and to supply an interface for the testbench to access the signals.



For example, a BFM might contain all the wires of an 'abc' bus, and a task called 'read(address, data)' which when called would issue a read on the bus, and return the data read from the address.

```
interface abc_if(input wire CLK);
  logic RST;
  logic VALID;
  logic READY;
  logic RW;
  logic BURST;
  logic [31:0] ADDR;
  logic [31:0] DATAI;
  wire[31:0] DATAAO;
  task      reset();
  task      read( bit[31:0] addr, output bit[31:0] data);
  task      write( bit[31:0] addr, input  bit[31:0] data);
  task burst_read( bit[31:0] addr, output bit[31:0] data[4]);
  task burst_write( bit[31:0] addr, input  bit[31:0] data[4]);
  task      monitor(output bit transaction_ok,
                    output bit rw,
                    output bit[31:0] addr,
                    output bit[31:0] data[4],
                    output bit burst);
endinterface
```

This 'read' task consumes time. All tests are written in terms of the BFM and the helper functions that it supplies. No direct pin wiggling is allowed. The tasks in the BFM represent the kinds of bus transactions that the interface supports; for example read(), write(), burst\_read() and burst\_write(). In order to create tests, the BFM tasks are called, and signals are never directly accessed outside the BFM.



The BFM is a Bus Functional Model. It is the way that the test interacts with the signals. The BFM is implemented as a SystemVerilog interface that has signals (wires) and tasks to manipulate the signals.

The tasks in the BFM represent the kinds of transactions that the interface supports; for example, `read()`, `write()`, `burst_read()` and `burst_write()`. In order to create tests, the BFM tasks are called, but signals are never directly accessed outside the BFM.

Organizing tests in this way - as a BFM that interacts with the DUT using tasks that drive and sample signals - also allows an easy path to creating UVM testbenches that can be accelerated using an emulation platform.

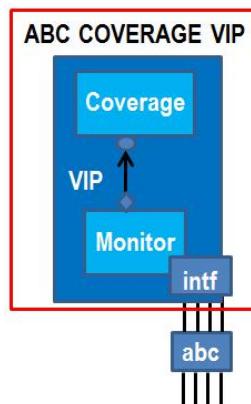
For more information, see [UVM Express - Writing BFM Tests](#)

## UVM Express: Functional Coverage

The second level of UVM Express adds a functional coverage to the BFM based tests.

Functional coverage is coverage that is concerned about the correct behavior of a device. For example, using a SystemVerilog covergroup, we can make sure that the device was used in a complete way, and that it created correct replies in each case. A memory might be read from the lowest address and the highest address, and a few addresses in between. Each read should produce correct values. A read to an illegal address should be handled correctly. Each of these inputs (READ addresses) and outputs (values READ) are counted, and can be used to ensure that all required functionality has been checked.

The functional coverage agent (a coverage collector) monitors the bus, creating transactions and publishing them to subscribers. A typical subscriber is a will collect coverage. Each published transaction gets counted by the covergroups in the coverage agent. Usually the coverage collector is written as a UVM Agent, and placed in a package. A user of this UVM agent does not need to interact directly with UVM, nor do they need to understand how the agent is built. They only need to understand any configuration options that the UVM agent offers. The UVM Agent is simply created and connected to the bus.



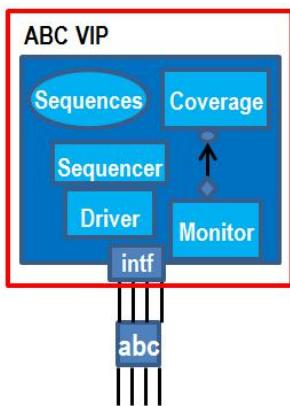
For more information, see [UVM Express Functional Coverage](#)

## UVM Express: Constrained Random Stimulus

The third level of UVM Express replaces the user directed test with constrained-random stimulus.

Constrained random stimulus is stimulus that has been described by using SystemVerilog constraints. The constraints are randomized to produce a new collection of stimulus. Each randomization begins with a seed value. A series of stimulus can start with a specific seed, and progress. Once the stimulus progression is done, a different constraint can be used with the same or different seeds to produce a different progression of stimulus.

User directed tests can be combined with constrained-random stimulus, but this third level of UVM Express doesn't attempt this. This level builds a UVM agent which can generate stimulus and collect coverage. This UVM agent is used in the same way as the UVM agent from the second level - it is created and connected. Once it is created, connected, configured and started, it causes transactions to occur on the bus - it is a traffic generator. Transactions are generated from the built-in randomization contained in the UVM agent - usually sequences and sequence items with constraints. A user of this UVM agent does not need to interact directly with the UVM or the UVM agent. The UVM agent is a self-contained stimulus generation tool. The user must understand what configuration options are available, but does not need to understand low level UVM details.



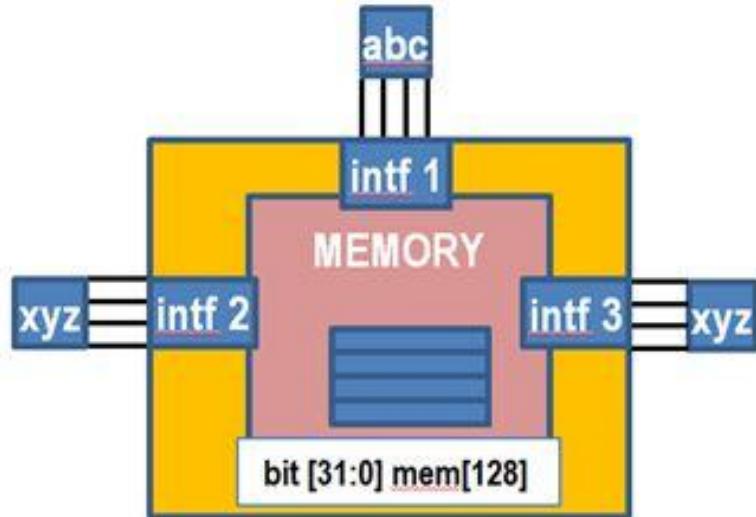
For more information, see UVM Express Constrained-Random Stimulus

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

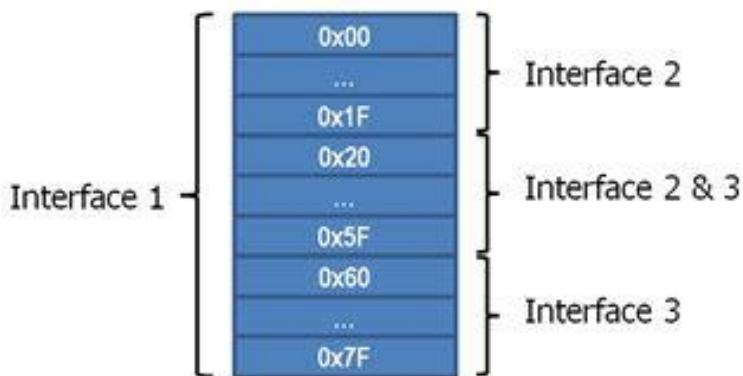
# UvmExpress/DUT

## DUT Interfaces

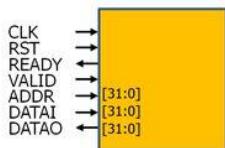
The example device-under-test that is used in the UVM Express examples has three interfaces. Those three interfaces are used to access a simple internal memory. Each interface can read and write the memory, but only one interface can be reading or writing at a time. Furthermore, the interfaces have address range restrictions.



Interface 2 is allowed to access addresses 0x00 to 0x5F, interface 3 is allowed to access addresses 0x20 to 0x7F. Interface 1 is allowed to access any legal address, from 0x00 to 0x7F.



Each interface contains the identical signals, but interface 2 and interface 3 are the same type, and interface 1 is different. The interface signals are listed below. They are either input (CLK, RST, VALID, ADDR, DATAI) or output (READY, DATAO). There are two widths - a single bit, or 32 bits.



## DUT Internals

### The memory

The memory on the DUT is a simple array of 32 bit values. There are a total of 128 values. The first 32 addresses are accessible by interface 1 and 2. The last 32 addresses are accessible by interface 1 and 3. The middle 64 addresses are accessible by all the interfaces.

```
// Address map
// Interface1&2 : 0- 31 (32 addresses)
// Shared       : 32 - 95 (64 addresses)
// Interface1&3 : 96 -127 (32 addresses)
reg[31:0] mem[128];
```

### Special Behavior

The DUT has some special behavior which is useful for our example. If the address 0xFFFFFFFF is written, or read, then the memory will dump its contents to standard output (the transcript). The DUT will go to sleep on a regular basis. Every 100 clocks, the DUT "sleeps" for 20 clocks, and then wakes up. While the DUT is sleeping, it pulls the READY signal low - meaning the DUT is not ready to accept any information. When the DUT is sleeping, each interface is "not ready". When the DUT is reset, the contents of the memory are set to 'X'. Any of the interfaces can independently reset the memory.

## Verilog Details

### DUT Connection Declaration

The dut module definition has the following connections:

```
module dut_mem(
  input  wire          CLK1,
  input  wire          RST1,
  input  wire          RW1,
  output reg          READY1,
  input  wire          VALID1,
  input  reg [31:0]    ADDR1,
  input  reg [31:0]    DATAI1,
  output reg [31:0]    DATAO1,
  input  wire          CLK2,
  input  wire          RST2,
  input  wire          RW2,
  output reg          READY2,
  input  wire          VALID2,
  input  reg [31:0]    ADDR2,
  input  reg [31:0]    DATAI2,
  output reg [31:0]    DATAO2,
```

```

input  wire      CLK3,
input  wire      RST3,
input  wire      RW3,
output reg      READY3,
input  wire      VALID3,
input  reg [31:0] ADDR3,
input  reg [31:0] DATAI3,
output reg [31:0] DATAO3
);

```

## DUT Instantiation Connection

When it is instantiated, it is instantiated like any other module instance.

```

dut_mem dut (      // DUT
    .CLK1(clk),
    .RST1(bfm.RST),
    .RW1(bfm.RW),
    .READY1(bfm.READY),
    .VALID1(bfm.VALID),
    .ADDR1(bfm.ADDR),
    .DATAI1(bfm.DATAI),
    .DATAO1(bfm.DATAO),

    .CLK2(clk),
    .RST2(bfm2.RST),
    .RW2(bfm2.RW),
    .READY2(bfm2.READY),
    .VALID2(bfm2.VALID),
    .ADDR2(bfm2.ADDR),
    .DATAI2(bfm2.DATAI),
    .DATAO2(bfm2.DATAO),

    .CLK3(clk),
    .RST3(bfm3.RST),
    .RW3(bfm3.RW),
    .READY3(bfm3.READY),
    .VALID3(bfm3.VALID),
    .ADDR3(bfm3.ADDR),
    .DATAI3(bfm3.DATAI),
    .DATAO3(bfm3.DATAO)
);

```

The connection to the DUT is made by selecting wires from the BFM (SystemVerilog interface), or other signals in the top (clk).

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# UvmExpress/BFM

The UVM Express needs a BFM - a Bus Functional Model. The BFM in UVM Express is implemented as a SystemVerilog interface that contains signals, tasks and functions. The tests that are written are not allowed any direct access to the signals, they must instead use tasks and functions through the BFM. The only way to interact with the signals and in turn the RTL design is by using tasks and functions in the BFM.

## A Bus Functional Model

Bus Functional Models have been used for many years. They provide a simple, but powerful abstraction for access to the design under test. The first step to using UVM Express is to build a collection of tasks and functions that encompass the way the DUT is used. For example, a simple BFM might have read() and write() tasks. By calling the read() task, a read would occur on the bus or interface of the design under test.

```
task read(bit[31:0] addr, output bit[31:0] data);
```

```
    RW <= 1;          // Going to be a READ.
    BURST <= 0;      // Not a burst.
    ADDR <= addr;   // Put out the address.
    VALID <= 1;      // Signal valid...
    DATAI <= 'z;
    while (!READY) wait(READY); // Wait for response ready.
```

```
    @(posedge CLK);
    data = DATAO; // Get the response
```

```
    $display("      : %m READ (%0x)", data);
    VALID <= 0;      // Not valid any more.
endtask
```

A burst command may also be available with this interface, and DUT.

```
task burst_read(bit[31:0] addr, output bit[31:0] data[4]);
```

```
    RW <= 1;          // Going to be a READ.
    BURST <= 1;      // Burst yes.
    DATAI <= 'z;
```

```
    for(int i = 0; i < 4; i++) begin
        ADDR <= addr+i; // Update the burst address.
        VALID <= 1;      // Signal a valid transfer.
        while (!READY) wait(READY);
        @(negedge CLK);
        @(posedge CLK);
        data[i] = DATAO;// Get the response.
```

```

    end

    VALID <= 0; // All done.

    $display("@ %05t: %m BURST READ (%0x, %0x, %0x, %0x, %0x)",
        $time, addr, data[0], data[1], data[2], data[3]);
endtask

```

Each BFM or interface should have a monitor. This monitor is responsible for recognizing traffic on the bus or interface. A monitor can be quite tricky to write, but is a necessary step for analysis and debug. In the monitor below, all the arguments to the monitor task are output variables - they are returned values when the monitor returns. Each time the monitor is called, it returns one "transaction". This monitor has two kinds of transactions it can return - a single transfer and a burst transfer. The monitor will be active for either one clock, or 4 clocks, depending on whether a single transfer or a burst transfer is being recognized.

```

task monitor(output bit transaction_ok,
            bit rw,
            bit[31:0]addr,
            bit[31:0]data[4],
            bit burst);

    transaction_ok = 0;
    @(posedge CLK);
    if ((READY == '1) && (VALID == '1) && (BURST == '1)) begin
        int burst_count;

        burst = 1;
        rw = RW;
        addr = ADDR;

        burst_count = 0;
        while ((READY == '1) && (VALID == '1) && (BURST == '1)) begin
            @(posedge CLK);
            if (rw == 1) // READ
                data[burst_count] = DATA0;
            else // WRITE
                data[burst_count] = DATA1;
            burst_count++;
            if (burst_count >= 4)
                break;
        end
        if (burst_count != 4)
            $display( "MONITOR ERROR: Burst error - %0d should have been 4", burst_count);
        transaction_ok = 1;
    end
    else if ((READY == '1) && (VALID == '1) && (BURST == '0)) begin
        burst = 0;
    end

```

```

rw = RW;
addr = ADDR;
@(posedge CLK);
if (rw == 1) // READ
  data[0] = DATA0;
else          // WRITE
  data[0] = DATA1;
transaction_ok = 1;
end
endtask

```

## A SystemVerilog Interface

### Building the interface

The BFM should contain all the ways that a test or a monitor is going to interact with the device under test. An example interface skeleton used in the example is below. The BFM or interface to the device under test is implemented using a SystemVerilog 'interface' construct. An interface can be thought of as a special kind of module which is instantiated and connected like a module. UVM Express will use the interface in a simple way, calling tasks and functions using the interface.

```

interface abc_if(input wire CLK);

  logic RST;

  logic VALID;
  logic READY;

  logic RW;
  logic BURST;

  logic [31:0] ADDR;
  logic [31:0] DATA1;
  wire[31:0] DATA0;

  task reset();
  ...
endtask

task read(bit[31:0] addr, output bit[31:0] data);
  ...
endtask

task write(bit[31:0] addr, input bit[31:0] data);
  ...

```

```

  endtask

  task burst_read(bit[31:0] addr, output bit[31:0] data[4]);
    ...
  endtask

  task burst_write(bit[31:0] addr, bit[31:0] data[4]);
    ...
  endtask

  task monitor(output bit transaction_ok,
               bit rw,
               bit[31:0]addr,
               bit[31:0]data[4],
               bit burst);
    ...
  endtask
endinterface

```

## Using the interface

The SystemVerilog interface is instantiated, and any connections are provided. Interface 'abc\_if' is instantiated below using an instance name of 'bfm', and is connected to a signal called 'clk'. This is just a normal kind of module instantiation and connection.

```
abc_if bfm(clk); // BFM - the Bus Functional Model
```

Once the interface is instantiated, it can be connected to the device under test.

```

module top();
  reg clk;

  abc_if bfm(clk); // BFM - the Bus Functional Model
  ...

  dut_mem dut (      // DUT
    .CLK1(clk),
    .RST1(bfm.RST),
    .RW1(bfm.RW),
    .READY1(bfm.READY),
    .VALID1(bfm.VALID),
    .ADDR1(bfm.ADDR),
    .DATA11(bfm.DATA1),
    .DATA01(bfm.DATA0),
  ...

```

```
 );
endmodule
```

A simple test can be written using the reset(), read() and write() routines in the BFM. This test first resets the BFM (and in turn, the device under test). Then, the test issues a write to address 10, using the data value 42. Next a read is issued, and the value read is returned in the variable 'rdata'. Finally, rdata and 42 are compared to see that what was written was the same as what was read.

```
initial begin
    int rdata;
    top.bfm.reset();

    top.bfm.write(10, 42);      // Write
    top.bfm.read( 10, rdata); // Read

    if ( rdata != 42 )
        $display("ERROR: Data mismatch @%0x (expected %0x, got %0x)",
                  10, 42, rdata);
end
```

## Transaction based Monitor

The BFM-based monitor task above recognizes one transaction and then returns, so it should be called in a loop. The BFM monitor should be called after each transaction is recognized. For example, in a transaction based monitor, the BFM monitor task could be called, returning the information from the bus transaction. That returned information is put into a transaction, and sent to any subscriber on the analysis port, 'ap'. A transaction based monitor is a "nice-to-have" for UVM Express Step 1, but is a "must-have" for UVM Express Steps 2 and 3 (Functional Coverage and Constrained-Random Stimulus).

```
forever begin
    // Each call to monitor waits for at least one clock
    bfm.monitor(transaction_ok, rw, addr, data, burst);
    // Handle the recognized transaction
    ...
    ap.write(t);
end
```

( **download source code examples online at <http://verificationacademy.com/uvm-ovm> ).**

# UvmExpress/WritingBfmTests

The first tests that might be written using UVM Express and the BFM could be quite simple. They are likely going to be directed tests. There might be some randomization of data or addresses, or packet contents. There might be some randomness about the order of instructions that are being sent to an interface. These kinds of tests may lead to the need to specify relationships between data or packets. For example, constraining the packet to have an odd checksum, at the same time header and payload are constrained to have certain values and data lengths. These kind of random tests are best written using the full power of the UVM and with SystemVerilog functional coverage. These constrained random kinds of tests can be included in the last stop in UVM Express, Constrained-Random Stimulus, but are not considered in this section, Writing BFM Tests.

The tests described here are simple directed tests using a BFM with an API to generate interesting stimulus. Eventually, tests may be written that utilize Verification IP and/or the full UVM and constrained random stimulus generation. This chapter will describe the simple BFM tests, combined with some parts of SystemVerilog.

## Writing a first test

We've already written a first test in the previous chapter.

```
initial begin
    int rdata;
    top.bfm.reset();

    top.bfm.write(10, 42);      // Write
    top.bfm.read( 10, rdata); // Read

    if ( rdata != 42 )
        $display("ERROR: Data mismatch @%0x (expected %0x, got %0x)",
                  addr, 42, rdata);
end
```

In this test, we used the BFM to call three routines. We write one piece of data, and read one piece of data, comparing the written and read data. This is a perfectly good test, but doesn't test very much. In our example, we know that there are 128 addresses. We could write a test that writes to each address, and then reads the value back:

```
for (int i = 0; i < 128; i++) begin
    bfm.write(i, i);
end

for (int i = 0; i < 128; i++) begin
    bfm.read(i, rdata);
    if (rdata != i)
        // ...Handle the Error...
end
```

Instead of writing all 128 values, and then reading them back, we could write and read each address in turn:

```

for (int i = 0; i < 128; i++) begin
    bfm.write(i, i);
    bfm.read(i, rdata);
    if (rdata != i)
        // ...Handle the Error...
end

```

Maybe we want to test any aliasing between adjacent addresses. First we write an address. Then we write a 0 word below and above. Then we read the original address, and check that the value is good. Repeat for the value all ones - above and below.

```

for (int i = 0; i < 128; i++) begin
    bfm.write(i, i);
    bfm.write(i-1, '0);
    bfm.write(i+1, '0);
    bfm.read(i, rdata);
    if (rdata != i)
        // ...Handle the Error...

    bfm.write(i-1, '1);
    bfm.write(i+1, '1);
    bfm.read(i, rdata);
    if (rdata != i)
        // ...Handle the Error...
end

```

The power of SystemVerilog is available to write tests. If you wanted to write to interface 3, and read from interface 2, you could write a test like:

```

bfm3.write(10, 42);
bfm2.read(10, rdata);
if (rdata != 42) begin
    // Handle error
end

```

We might want to write to all three interfaces at once. We could create three "threads" using initial or always blocks, or we could use fork/join. In either case, the write() function calls are being called in parallel, and we're testing the ability of the device under test to cope with simultaneous write requests.

## Initial blocks

```

initial begin
    bfm.write(10, 42);
end
initial begin
    bfm2.write(10, 43);
end

```

```
initial begin
  bfm3.write(10, 44);
end
```

## Fork/Join

```
fork
  bfm.write(10, 42);
  bfm2.write(10, 43);
  bfm3.write(10, 44);
join
```

## Libraries of tests

Now that we have the device under test, the BFM, a BFM-based API, we can write some tests using that API. Those tests are usually written as a collection of instructions contained in a task. The task becomes a "super-test", and a collection of these tasks becomes a library of tests.

For example, our write-all-addresses test from above can be built as a task:

```
task write_all_addresses();
  for (int i = 0; i < 128; i++) begin
    bfm.write(i, i);
  end
  for (int i = 0; i < 128; i++) begin
    bfm.read(i, rdata);
    if (rdata != i)
      // ...Handle the Error...
  end
endtask
```

Another test can call 'write\_all\_addresses'. This builds another layer into our test library. The library of tests is a very useful idea.

The UVM has a kind of stimulus generation class called a 'sequence'. Sequences in the UVM can be thought of as collections of tests, but are not really simple task libraries. They are much more powerful, but the idea is the same. The library of tests is a library of tasks with each task performing a useful testing function. Tests can be combined together in higher level tasks. When you use sequences, you create a library of sequences with each sequence performing a useful testing function. Sequences can be combined together in higher level sequences.

Running a collection of tasks will cause useful stimulus.

Running a collection of sequences will cause useful stimulus. Sequences are not in use at this level of the UVM Express, but you should keep them in mind as a way to write tests that is similar to writing tasks, but more powerful.

## Simulation

Simulation will require the device under test, the BFM, and any test.

### A test module (tests.sv)

The tests are contained in a module named 'tests'. It will be used as a top-level module when simulation is run. These tests are kept simple for this example, and depend on the BFMs hierarchical path being 'top.bfm', 'top.bfm2' and 'top.bfm3'. Later we'll see how to write tests that don't have this kind of requirement.

The tests module contains Verilog code - in this case an 'initial' block that invokes the tests desired. To run different tests, we could create a new file (tests1.sv) containing a tests or tests1 module. Then compile the new test and re-simulate with that new test module.

The test defines the entire simulation run. We start and end in the initial block. Any special startup, checking or shutdown code should be invoked here. In our example, the only shutdown code is a simple call to \$finish(2).

The test below uses two tasks. The first task (test1) does simple writes and reads to selected addresses. The second task (test2) does burst tests. The two tests run sequentially, and when they are finished, the simulation is done.

```
module tests();

initial begin
    test1(); // Run a test.
    test2(); // Another test. Burst.
    $finish(2);
end

task test1();
    int rdata, wdata;
    $display("\nTest 1 - reset, write/read");

    top.bfm.reset();

    for(int addr = 40; addr < 50; addr++) begin
        wdata = addr;
        top.bfm.write(addr, wdata); // Write
        top.bfm.read( addr, rdata); // Read

        if ( wdata != rdata )
            $display("ERROR: Data mismatch @%0x (expected %0x, got %0x)",
                     addr, wdata, rdata);
    end
    top.dut.dump();
endtask

task test2();
    bit [31:0]wdata_array[4] = {'h04, 'h03, 'h02, 'h01};

```

```

bit [31:0]xdata_array[4] = {'hf4, 'hf3, 'hf2, 'hf1};
bit [31:0]rdata_array[4];
$display("\nTest 2 - reset, burst_write/burst_read");

top.bfm.reset();

for (int start_addr = 'h10;
      start_addr <= 'h70;
      start_addr += 'h10) begin
  top.bfm.burst_write(start_addr, wdata_array);
  top.bfm.burst_write(start_addr+8, xdata_array);

  rdata_array = '0, 0, 0, 0;
  top.bfm.burst_read( start_addr, rdata_array);

  foreach(wdata_array[i])
    if ( wdata_array[i] != rdata_array[i] )
$display("ERROR: Data mismatch @%0x, element %0x (expected %0x, got %0x)",
        start_addr+i, i, wdata_array[i], rdata_array[i]);
  end
  top.dut.dump();
endtask
endmodule

```

## A top-level (top.sv)

The top level file contains a top module named 'top'. Inside top the BFM are instantiated, along with the device under test. The BFM and the DUT are connected together, and any clock or reset generation is done here.

```

module top();
  reg clk;

  abc_if bfm(clk); // BFM - the Bus Functional Model
  xyz_if bfm2(clk); // BFM 2 - NOT USED
  xyz_if bfm3(clk); // BFM 3 - NOT USED

  dut_mem dut( // DUT
    .CLK1(clk),
    .RST1(bfm.RST),
    .RW1(bfm.RW),
    .READY1(bfm.READY),
    .VALID1(bfm.VALID),
    .ADDR1(bfm.ADDR),
    .DATA1(bfm.DATA1),
    .DATA01(bfm.DATA0),

```

```
// BFM 2 and BFM 3 are not used in this test.

.CLK2(clk),
.RST2(bfm2.RST),
.RW2(bfm2.RW),
.READY2(bfm2.READY),
.VALID2(bfm2.VALID),
.ADDR2(bfm2.ADDR),
.DATAI2(bfm2.DATAI),
.DATAO2(bfm2.DATAO),

.CLK3(clk),
.RST3(bfm3.RST),
.RW3(bfm3.RW),
.READY3(bfm3.READY),
.VALID3(bfm3.VALID),
.ADDR3(bfm3.ADDR),
.DATAI3(bfm3.DATAI),
.DATAO3(bfm3.DATAO)

);

// Turn on the clock.

always begin
#10 clk <= 1;
#10 clk <= 0;
end

endmodule
```

## ABC BFM ( abc\_bfm/abc\_if.svh )

```
interface abc_if(input wire CLK);

logic RST;
logic VALID;
logic READY;
logic RW;
logic BURST;
logic [31:0] ADDR;
logic [31:0] DATAI;
wire [31:0] DATAO;

task reset();
...
endtask
```

```

task read(bit[31:0] addr, output bit[31:0] data);
  ...
endtask

task write(bit[31:0] addr, input bit[31:0] data);
  ...
endtask

task burst_read(bit[31:0] addr, output bit[31:0] data[4]);
  ...
endtask

task burst_write(bit[31:0] addr, bit[31:0]data[4]);
  ...
endtask

task monitor(output bit transaction_ok,
             bit rw,
             bit[31:0]addr,
             bit[31:0]data[4],
             bit burst);
  ...
endtask
endinterface

```

## Compiling and Simulating

There are many ways to arrange the top, tests and BFM. They can be compiled in many different ways, and simulation can be invoked many different ways. For our simple tests and simple DUT we're keeping things simple. We compile each file individually, and run simulation with two tops. The tests that we wrote depend on the BFM being known as 'top.bfm', so changing the instance hierarchy is going to break our tests. There are many solutions to this issue discussed later, but this technique has been used for years, and is an easy use-model for this simple example.

```

rm -rf work
vlib work
vlog  ./dut/rtl.sv          # Compile the DUT
vlog top.sv                  # Compile the TOP
vlog ../abc_bfm/abc_if.svh   # Compile the ABC interface (BFM)
vlog ../xyz_bfm/xyz_if.svh   # Compile the XYZ interface (BFM)
vlog tests.sv                 # Compile the tests

vsim top tests               # Simulate with two tops - top and tests

```

To compile other tests, and run those, you might **do**:

```
vlog tests2.sv          # Compile the tests
vsim top tests2          # Simulate with two tops - top and tests
```

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

# UvmExpress/FunctionalCoverage

Writing tests means providing input stimulus that causes observable behavior. Many times input stimulus is applied, and expected output is compared with actual output. This way functional behavior can be verified. Test writing is an important part of verification, as is checking expected results versus actual results. SystemVerilog adds another tool to help with functional verification - functional coverage.

There are many kinds of coverage - line coverage, FSM coverage, protocol coverage, assertion coverage, functional coverage. Line coverage tells you if you executed each line. FSM coverage tells you if you have visited every state and taken every transition. Protocol coverage tells you if you have exercised all the variations in protocol - like ready-before-valid, valid-before-ready, reads and writes, or burst reads of different sizes. Assertion coverage tells you if your assertions have been covered. Those assertions can be checks for any condition. Functional coverage tells you if a certain condition has existed, and how many times it has existed. For example, using functional coverage, you can count how many small, medium or large packets have been transferred, and if there were conditions where a large packet was transferred on one interface while a small packet was transferred on another interface. Functional coverage can be used to ensure that desired functionality has been exercised or that undesired functionality has not occurred. Functional coverage is described by using SystemVerilog covergroups.

## Using Covergroups

SystemVerilog covergroups are very powerful. They are counters that count "sampled" data. When an interesting piece of data is to be sampled, the covergroup sample() function is called.

```
cg.sample();
```

This causes the covergroup to evaluate whether to count the sample or not, and where to put the count.

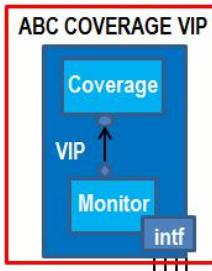
The covergroup below is a simple covergroup that samples 3 coverpoints - burst, rw and addr. The values sampled are members of the class 't', which is of type 'abc\_item'. Additionally, the covergroup defines 2 'crosses'. Crosses collect correlated coverage - what are the two values when sample() is called. For example 'burst\_X\_rw' is a cross between burst and rw. Both burst and rw are bits. The cross of burst and rw create 4 things that need counting - burst read, burst write, non-burst read and non-burst write (the four possible combinations).

```
abc_item t;

covergroup cg;
    burst_cp: coverpoint t.burst;
    rw_cp:    coverpoint t.rw;
    addr_cp:  coverpoint t.addr;
    burst_X_rw: cross burst_cp, rw_cp;
    addr_X_rw: cross  addr_cp, rw_cp;
endgroup
```

## Monitoring and sampling

Writing good coverage can be quite hard. It is hard to describe all the conditions (the complete functionality that you want to cover). One quick way to get good coverage is to use coverage that someone else has built. Whether you build it yourself or use a third party coverage collector, the verification environment will look the same. The coverage collector gets a transaction that was created from the bus monitor, and samples the covergroup.



The ABC Coverage VIP above, a UVM Agent, contains a TLM-monitor, a coverage collector and a transaction that gets sent from the monitor to the coverage collector. The TLM-monitor in the VIP uses a BFM monitor task to observe the interface, recognizing transactions. When a transaction is identified, it is sent to the coverage collector.

The run\_phase of our ABC Monitor is listed below. It uses the BFM monitor task to capture a transaction. Once the BFM monitor returns, the transaction is sent to the coverage collector using 'ap.write(t)'. This monitor runs forever, calling the BFM monitor, forwarding a transaction and repeating.

```

task run_phase(uvm_phase phase);
    abc_item t;

    if (bfm == null) begin
        `uvm_fatal("MONITOR", "No BFM or virtual interface")
    end

    t = abc_item::type_id::create("t");
    forever begin
        bit transaction_ok;

        // Each call to monitor waits for at least one clock
        bfm.monitor(transaction_ok,
                    t.rw, t.addr, t.data, t.burst);
        if (transaction_ok) begin
            `uvm_info("MONITOR",
                      $sformatf(" %s (%s)",
                                (t.burst==1)?" Burst": "Normal", t.convert2string()),
                      UVM_MEDIUM)
            ap.write(t);
            t = abc_item::type_id::create("t");
        end
    end
endtask

```

## Adding Coverage using a VIP

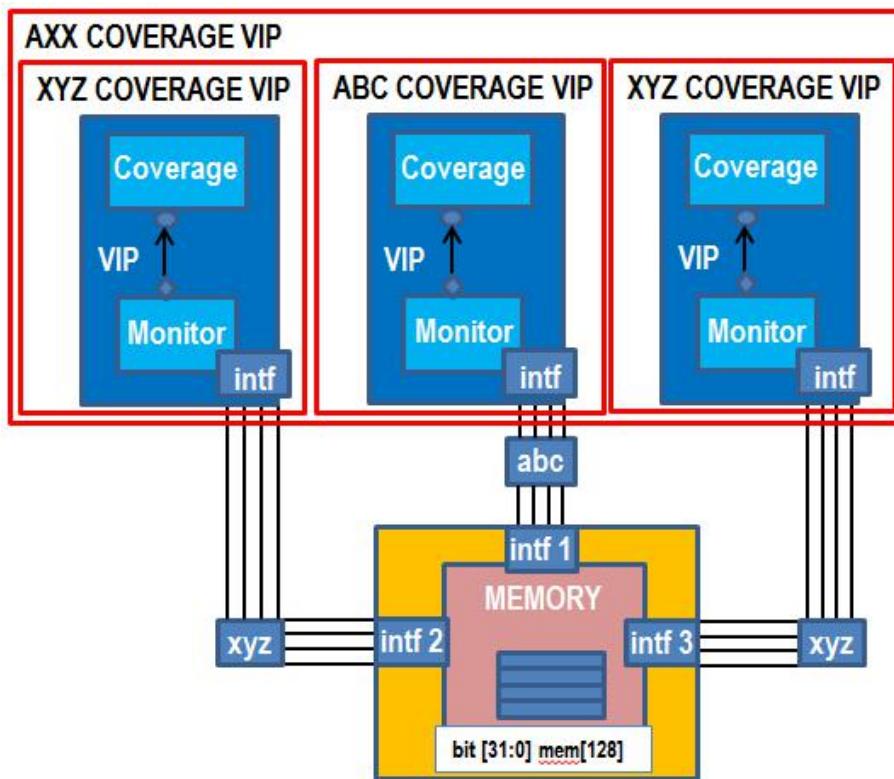
Adding coverage to our previous task based BFM test bench is easy. We're going to connect the coverage VIP to the BFM. The BFM is a SystemVerilog interface, and the coverage VIP wants to be connected to a SystemVerilog interface.

This Coverage VIP is written using the UVM, but we don't have to know too much about the UVM in order to use the VIP.

The writer of the coverage VIP must be able to write covergroups, and must understand how the UVM will be connected, and how to design transactions.

The user of the coverage VIP needs to know what configuration parameters are available, and how to connect it and instantiate it, but he does not need to know any UVM details or any protocol specifics. The VIP encapsulates the protocol and the UVM implementation.

In this example, we are using a VIP which contains three interfaces (and three coverage collectors). The VIP is built up of three lower level VIPs (an ABC VIP and two XYZ VIPs).



## Instantiating and "connecting" the coverage collector

In order to use this coverage collector, we'll need to change our original test bench. Below, the left hand code block contains our original test. It is a top-level module that starts two tests, and exits. The right hand code block contains the original test with functional coverage added.

The new code uses import to bring in the UVM code, and the actual VIP. Next it registers the three interfaces - top.bfm1, top.bfm2 and top.bfm3 with the VIP. Finally, it uses 'run\_test()' to start the VIP test, named "axx\_coverage\_test". It is not really a test, but it is a component in the VIP which will create the monitor and coverage collector and connect them. Each monitor (three will be running - one for each interface being tested) runs independently and will begin creating transactions immediately.

**Original BFM Task based tests**

```
module tests();
  // Tests start here.
  initial begin
    test1(); // Run a test.
    test2(); // Another test. Burst.
    $finish(2);
  end

```

**Original BFM Task based tests with Functional Coverage added**

```
module tests();
  import uvm_pkg::*;
  import axx_coverage_pkg::*;
  // Bring in run_test().
  // Bring in the AXX
  // Coverage VIP.

  initial begin
    axx_coverage_test::register_abc("bfm1", top.bfm1);
    axx_coverage_test::register_xyz("bfm2", top.bfm2);
    axx_coverage_test::register_xyz("bfm3", top.bfm3);
    uvm_config_db#(int)::set(uvm_root::get(), "*", "recording_detail", 1);
    run_test("axx_coverage_test");
  end

  // Tests start here.
  initial begin
    test1(); // Run a test.
    test2(); // Another test. Burst.
    //$/finish(2);
    // Cause UVM phasing to complete.
    axx_coverage_test::ready_to_finish();
  end

```

The VIP needs to be started using `run_test()`. When the test is complete, it needs to tell the VIP that it has finished using `axx_coverage_test::ready_to_finish()`. Once `axx_coverage_test::ready_to_finish()` is called, the VIP will cleanup, and cause an orderly shutdown of the VIP processing.

Note: The funny looking function calls

```
axx_coverage_test::register_abc("bfm1", top.bfm1);
axx_coverage_test::register_xyz("bfm2", top.bfm2);
axx_coverage_test::register_xyz("bfm3", top.bfm3);
uvm_config_db#(int)::set(uvm_root::get(), "*", "recording_detail", 1);
```

are just that - funny looking function calls. The first one is a function call which is used to register the BFM for the ABC interface in our test. The second two are function calls which are used to register the BFM for the two XYZ interfaces in our test. Each of these calls takes two arguments - a simple name ("bfm1", "bfm2", "bfm3"), and a BFM handle - a SystemVerilog virtual interface. These simple function calls have one line implementations, just calling into the UVM Configuration database to register the pair of values - ("bfm1" and top.bfm1 for example).

The last function call is quite difficult looking, but what it does is set "recording\_detail" to have the value 1 at the top of the UVM component hierarchy. This is an optional configuration step - not required to use the UVM Express. This configuration step effectively turns on transaction recording for the entire testbench (unless someone else explicitly turns it off). In the old OVM code, this call used to be `'set_config_int("*", "recording_detail", 1)'`.

The steps to use a coverage collector are simple. First import the uvm, and the coverage collector packages.

```
import uvm_pkg::*;
import axx_coverage_pkg::*;
// Bring in run_test().
// Bring in the AXX
// Coverage VIP.
```

In an initial block, register the BFM<sub>s</sub>, using the registration functions that are provided by the coverage collector (register\_abc() and register\_xyz() here). After the BFM<sub>s</sub> have been registered, the UVM phase engine is started using the UVM task 'run\_test'. The "test" that is started ("axx\_coverage\_test" here) is not really a test, but it simply constructs and connects the coverage collector, starting any necessary processes.

```
initial begin
    axx_coverage_test::register_abc("bfm1", top.bfm1);
    axx_coverage_test::register_xyz("bfm2", top.bfm2);
    axx_coverage_test::register_xyz("bfm3", top.bfm3);

    run_test("axx_coverage_test");
end
```

One final change is that the test should not cause the simulation to exit. The UVM may create final reports or other statistics as part of its final phases. If \$finish() is called too early, the UVM will not have a chance to produce these reports. The best recommendation is that you should not have any code which causes an exit, except the code that is in the UVM. Let the UVM finish the simulation.

Any \$finish() statements should be removed from the test. After all the tests have run, the coverage collector should be signaled that it is time to end simulation, by calling the provided 'ready\_to\_finish()' call.

```
// Tests start here.

initial begin
    test1(); // Run a test.
    test2(); // Another test. Burst.

    // Cause UVM phasing to complete.
    axx_coverage_test::ready_to_finish();
end
```

In order to move from a simple task-based BFM with no coverage to a task-based BFM with three sets of functional coverage, we simply added the 7 lines outlined above.

( **download source code examples online at <http://verificationacademy.com/uvm-ovm>** ).

# UvmExpress/ConstrainedRandom

Test writing using tasks and functions and collecting coverage can exercise a device under test, and can provide clues about functionality that hasn't yet been verified. Test writing using tasks and functions usually devolves into a directed test kind of test writing. For each thing being tested, a specific test is written. This process is time consuming, tedious and error prone. Randomization of data and control can be added to improve coverage, but those randomizations may not be able to capture relationships between data. Using constraints, capturing data relationships is easy.

Instead of writing a test per functional verification, you write a collection of constraints. The constraints are randomized to yield a solution from the set of all possible legal solutions. You use constrained random stimulus generation to cause many different sets of stimulus to be generated automatically. By changing constraints, the stimulus will change. Each constraint applies a different set of rules for the system. Additionally, once a set of constraints are defined, simulation can produce many different results, by varying the random seeds for the random number generation.

Eventually a collection of constraints is run multiple times, using different seeds. Certain seeds and constraints are found to produce a desired result. Those seeds and constraints define a collection of tests. The total of all constraints and seeds becomes the verification environment for the block or system.

## Using Constrained-Random Stimulus Generation

A SystemVerilog constrained-random stimulus generator is created by using constraints. Constraints are written describing legal or desired values and relationships between those values. The constraints are randomized, which selects a legal collection of values from the possible solution set.

The constraints below describe two simple constraints. The address must be divisible by 4 and in the range 0 to 127, and the data must be in the range 0 to 127.

```
constraint val_addr {
    addr[1:0] == 'b00;
    addr > 0;
    addr < 127; }

constraint val_data {
    data[0] > 0;
    data[0] < 127; }
```

If the transaction below (abc\_item) was randomized, a random address and data would be picked that satisfied the constraint. The class variables 'rw' and 'burst' are not constrained, so can take any legal value (0 or 1).

```
class abc_item extends uvm_sequence_item;
`uvm_object_utils(abc_item)

rand bit burst;
rand bit rw;
rand bit[31:0] addr;
rand bit[31:0] data[4];

constraint val_addr { // address divisible by 4, [0-127]
```

```

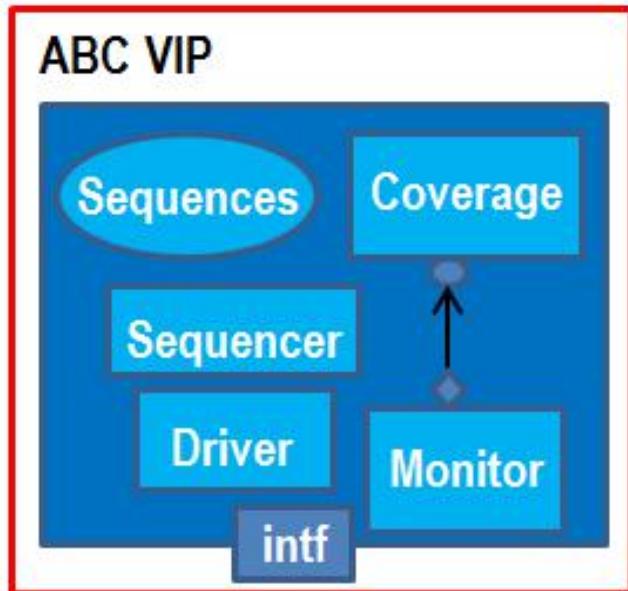
addr[1:0] == 'b00; addr > 0; addr < 127; }

constraint val_data { data[0] > 0; data[0] < 127; ... }

endclass

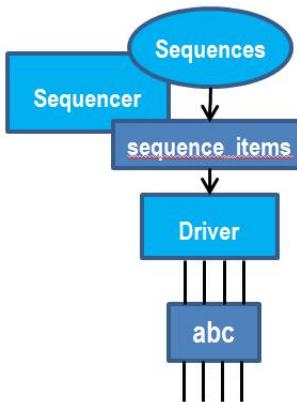
```

The result of randomizing the abc\_item above would create a transaction with a constrained address and data. It would be either a READ or a WRITE, and either a BURST or non-BURST.



## Adding Constrained-Random Stimulus using a VIP

The diagram below shows a sequence running on a sequencer. The sequence is creating a sequence\_item, and sending it to a driver (using the sequencer arbitration). The driver receives that sequence\_item, and acts upon it.



The sequence is generating random transactions. This means that the driver will see a random stream of transaction kinds - READ, WRITE, BURST\_READ or BURST\_WRITE.

The sequence is implemented as a task named 'body()'.

```

task body();
  abc_item item;
  abc_reset_item reset_item;

```

```
// .. Contents of the body() task.  
  
endtask
```

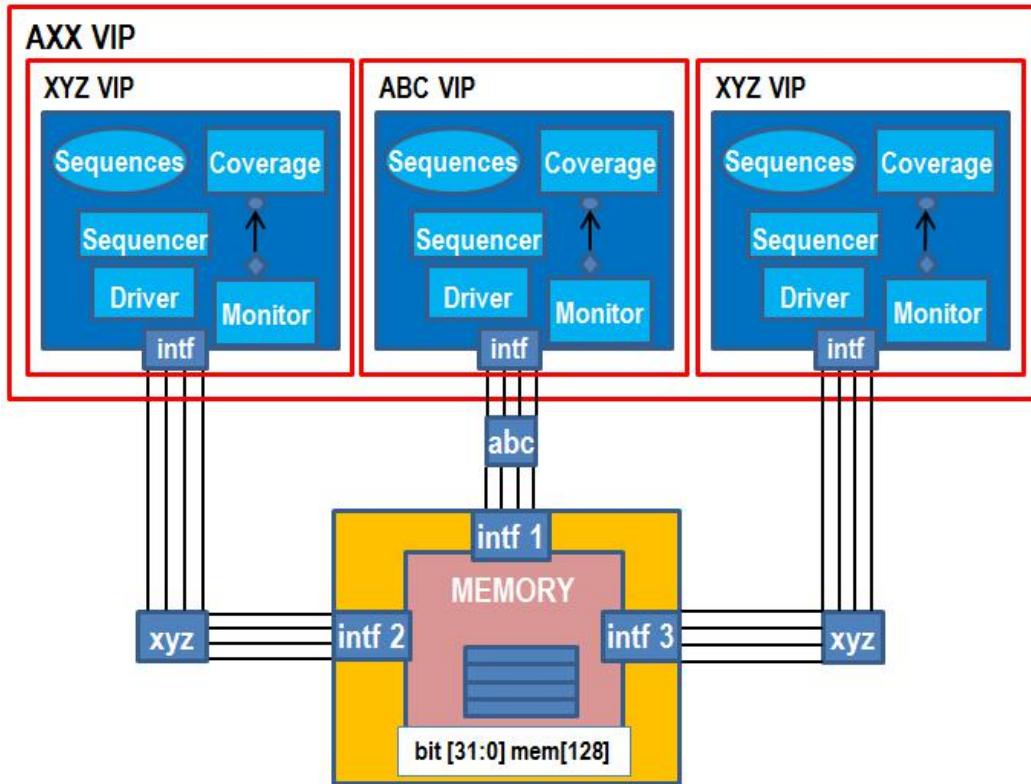
The body() task does two things in this sequence. It first creates a reset transaction using the factory, and then sends it to the driver using start\_item() and finish\_item().

```
// Reset the hardware  
  
reset_item = abc_reset_item::type_id::create("reset_item");  
start_item(reset_item);  
finish_item(reset_item);
```

Next, the body() enters an infinite loop. This loop will continuously create a new transaction, randomize it, and send it to the driver. The transaction type is abc\_item, and it is created using the factory function named 'create'. The transaction is randomized using 'item.randomize()'. This randomization is refined using the 'randomize() with ...' SystemVerilog construct. In this case, we are randomizing the transaction with the transaction address (addr) between the low address range limit (addr\_low) and the high address range limit (addr\_high). The addr\_low and addr\_high variables are variables in this sequence. This means that this while() loop will only generate transactions in the range addr\_low to addr\_high.

```
// Crank out reads and writes forever.  
  
while(1) begin  
    item = abc_item::type_id::create("item");  
  
    start_item(item);  
  
    assert(item.randomize() with { addr >= local::addr_low;  
                                addr <  local::addr_high;  
                                });  
    finish_item(item);  
end  
endtask
```

## Summary



## Using the AXX VIP

The verification IP above combines three instances of two different kinds of verification IP. There are two instances of the XYZ VIP and one instance of the ABC VIP. Together they form the AXX VIP (an ABC and two XYZ's). This combination VIP can be used to verify a device with three interfaces - one ABC and two XYZ.

Using the AXX VIP for constrained-random stimulus generation is easy. A technique similar to the technique applied for functional coverage is used. The UVM and AXX packages (uvm\_pkg and axx\_pkg) must be imported. Then the BFM's must each be registered. Next the test is configured, if needed. Finally, the test is begun a call to run\_test. The original directed tests are not reused in this test.

The AXX combination VIP is easy to use. The tests definition from the previous section is shown on the left. On the right is the test definition for the constrained-random stimulus. The earlier tests are not used for this example, but rather are replaced by the stimulus that is generated by the three VIPs the ABC VIP and the two XYZ VIPs.

Original Functional Coverage based tests

Original Functional Coverage based tests with  
Constrained-Random stimulus

```

module tests();
  import uvm_pkg::*;
  import axx_coverage_pkg::*;

  initial begin
    axx_coverage_test::register_abc("bfm1", top.bfm1);
    axx_coverage_test::register_xyz("bfm2", top.bfm2);
    axx_coverage_test::register_xyz("bfm3", top.bfm3);

    run_test("axx_coverage_test");
  end

  initial begin
    test1(); // Run a test.
    test2(); // Another test. Burst.
    //$/finish(2);
    // Cause UVM phasing to complete.
    axx_coverage_test::ready_to_finish();
  end

  task test1();
    ...
  endtask

  task test2();
    ...
  endtask
endmodule

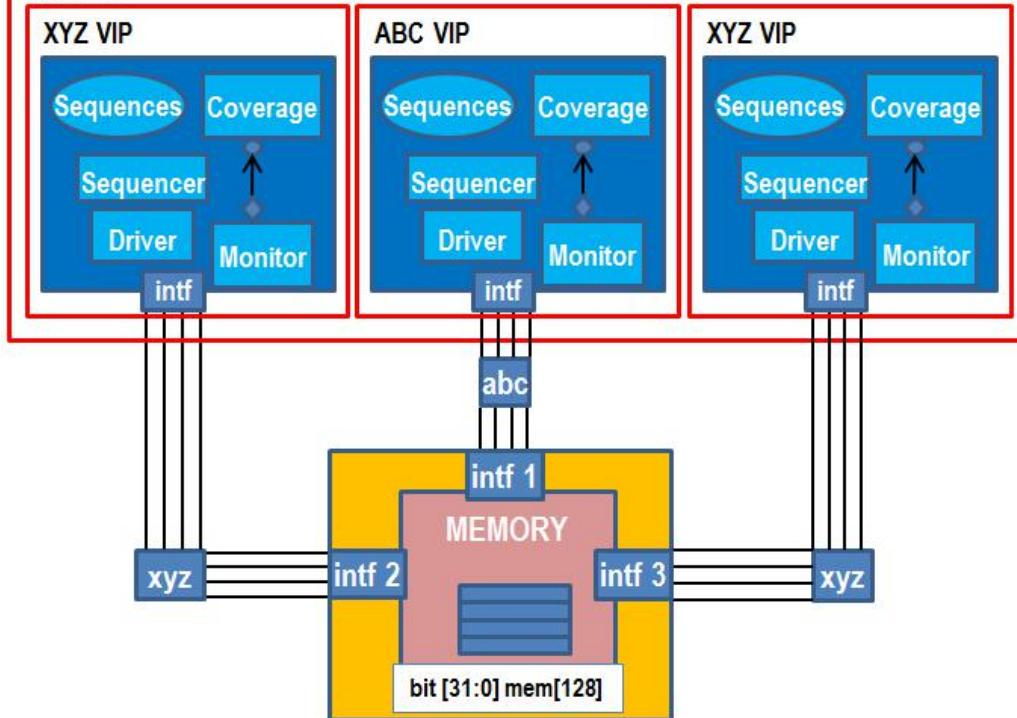
module tests();
  import uvm_pkg::*; // Bring in run_test().
  import axx_pkg::*; // Bring in the AXX VIP.

  initial begin
    axx_test::register_abc("bfm1", top.bfm1);
    axx_test::register_xyz("bfm2", top.bfm2);
    axx_test::register_xyz("bfm3", top.bfm3);

    run_test("axx_test");
  end
endmodule

```

### AXX VIP



## The AXX VIP

The AXX agent pulls the three sub-agents together. It is the combination VIP which can be used to verify our DUT with three interfaces.

### Construction and initialization

The `axx_test` class is the top-level test, and is responsible for building the testbench environment, and starting stimulus generation. The AXX VIP builds the sub-components - the sub-VIP, and then starts sequences on each interface.

The AXX test class definition contains three virtual interfaces, `bfm1`, `bfm2` and `bfm3`.

```
class axx_test extends uvm_env;
  `uvm_component_utils(axx_test)

  virtual abc_if bfm1;
  virtual xyz_if bfm2;
  virtual xyz_if bfm3;
```

These are handles to interfaces. Those interfaces may be declared in the top module. The interfaces are the way that the testbench communicates with the DUT.

The `axx_test` class also contains three handles to agents (`agent1`, `agent2` and `agent3`).

```
abc_agent agent1;
xyz_agent agent2;
xyz_agent agent3;
```

The agent handles are class handles to objects that will be created here in this test. They are the objects that will generate stimulus and collect functional coverage. Before we have constructed them, they have the value 'null'. After we construct them (using the `::create()` factory call in the `build_phase()` below), the handles will point to an appropriate object - a constructed object.

## Registering the BFM

The BFM are implemented as SystemVerilog interfaces. They are normally instantiated in the top level module. They must be communicated to the testbench using the configuration database. The functions '`register_abc`' and '`register_xyz`' are convenience functions that wrap a call to the `::set()` call for the configuration database. Calling these functions saves the virtual interface handles, so that they can be retrieved later. This is the way that BFM interfaces are "connected" to the testbench.

```
static function void register_abc(string name, virtual abc_if vif);
  uvm_config_db#(virtual abc_if)::set( uvm_root::get(), "", name, vif);
endfunction

static function void register_xyz(string name, virtual xyz_if vif);
  uvm_config_db#(virtual xyz_if)::set( uvm_root::get(), "", name, vif);
endfunction
```

## The build\_phase()

The AXX agent has three BFM handles, along with three class handles. The BFM handles are SystemVerilog interface handles that are stored in the configuration database (from the top level module). There is one BFM handle for each of the sub-agents. The complete build\_phase is listed below.

```
function void build_phase(uvm_phase phase);

  if( !uvm_config_db#(virtual abc_if)::get( this, "", "bfm1", bfm1))
    `uvm_fatal("ENV", "BFM1 not set")
  if( !uvm_config_db#(virtual xyz_if)::get( this, "", "bfm2", bfm2))
    `uvm_fatal("ENV", "BFM2 not set")
  if( !uvm_config_db#(virtual xyz_if)::get( this, "", "bfm3", bfm3))
    `uvm_fatal("ENV", "BFM3 not set")

  uvm_config_db#(virtual abc_if)::set( this, "agent1", "bfm", bfm1);
  uvm_config_db#(virtual xyz_if)::set( this, "agent2", "bfm", bfm2);
  uvm_config_db#(virtual xyz_if)::set( this, "agent3", "bfm", bfm3);

  agent1 = abc_agent::type_id::create("agent1", this);
  agent2 = xyz_agent::type_id::create("agent2", this);
  agent3 = xyz_agent::type_id::create("agent3", this);
endfunction
```

The BFM handles are retrieved using

```
uvm_config_db#(virtual abc_if)::get( this, "", "bfm1", bfm1);
```

The configuration database is used to retrieve a BFM handle with a key called "bfm1". That value was set in the top-level test. This lookup is repeated three times - one for each BFM handle. The AXX agent is a top-level agent, and it retrieved these handles from the top-level environment. It then passed them down to each of the individual agents using

```
uvm_config_db#(virtual abc_if)::set( this, "agent1", "bfm", bfm1);
```

This causes the configuration database to store a value of bfm1 with the key "bfm", for the agent, "agent1". Later, when agent1 is being built, it will do a configuration :: get() looking for the BFM handle registered with the name 'bfm'. This lookup will be specific to the agent, and so it will find the BFM handle for this agent.

The final job of the build\_phase() is to create the three agents using the factory:

```
agent1 = abc_agent::type_id::create("agent1", this);
agent2 = xyz_agent::type_id::create("agent2", this);
agent3 = xyz_agent::type_id::create("agent3", this);
```

It is important to create the agents after setting the configuration database.

## The run\_phase()

The run\_phase in the AXX agent is responsible for managing the stimulus generation for each of the three interfaces that are being tested.

The run\_phase() below creates three sequences, seq1, seq2 and seq3.

```
seq1 = abc_even_sequence::type_id::create("seq1");
seq2 =      xyz_sequence::type_id::create("seq2");
seq3 =      xyz_sequence::type_id::create("seq3");
```

Then the addr\_low and addr\_high are set for each one. Seq1 addresses will range from 0 to 'h60. Seq2 addresses with range from 'h20 to 'h80. Seq3 addresses will range across the full address space, 0 to 'h80.

```
seq1.addr_low = 'h00; seq1.addr_high = 'h60;
seq2.addr_low = 'h20; seq2.addr_high = 'h80;
seq3.addr_low = 'h00; seq3.addr_high = 'h80;
```

The run\_phase these raises an objection, and forks three sequences. When each of the sequences has completed, the fork/join will complete, and the objection will be dropped.

```
phase.raise_objection(this);
fork
  seq1.start(agent1.sequencer);
  seq2.start(agent2.sequencer);
  seq3.start(agent3.sequencer);
join
phase.drop_objection(this);
```

This will cause the UVM phasing engine to move to the next phase, and beginning the process of cleanup and ending simulation.

The fork/join of the sequences uses the sequence.start() to "start" a sequence on a sequencer. For example, seq1.start(agent1.sequencer) causes the sequence 'seq1' to be started or running on the sequencer of agent1. Sequence1 is causing stimulus to occur on agent1, which in turn is connected to BFM 1.

## Complete Listing of the axx\_test class

```
class axx_test extends uvm_env;
  `uvm_component_utils(axx_test)

  virtual abc_if bfm1;
  virtual xyz_if bfm2;
  virtual xyz_if bfm3;

  abc_agent agent1;
  xyz_agent agent2;
  xyz_agent agent3;

  static function void register_abc(string name, virtual abc_if vif);
```

```
uvm_config_db#(virtual abc_if)::set( uvm_root::get(), "", name, vif);
endfunction

static function void register_xyz(string name, virtual xyz_if vif);
  uvm_config_db#(virtual xyz_if)::set( uvm_root::get(), "", name, vif);
endfunction

function void build_phase(uvm_phase phase);

  if( !uvm_config_db#(virtual abc_if)::get( this, "", "bfm1", bfm1))
    `uvm_fatal("ENV", "BFM1 not set")
  if( !uvm_config_db#(virtual xyz_if)::get( this, "", "bfm2", bfm2))
    `uvm_fatal("ENV", "BFM2 not set")
  if( !uvm_config_db#(virtual xyz_if)::get( this, "", "bfm3", bfm3))
    `uvm_fatal("ENV", "BFM3 not set")

  uvm_config_db#(virtual abc_if)::set( this, "agent1", "bfm", bfm1);
  uvm_config_db#(virtual xyz_if)::set( this, "agent2", "bfm", bfm2);
  uvm_config_db#(virtual xyz_if)::set( this, "agent3", "bfm", bfm3);

  agent1 = abc_agent::type_id::create("agent1", this);
  agent2 = xyz_agent::type_id::create("agent2", this);
  agent3 = xyz_agent::type_id::create("agent3", this);
endfunction

task run_phase(uvm_phase phase);
  abc_even_sequence seq1;
  xyz_sequence      seq2, seq3;

  seq1 = abc_even_sequence::type_id::create("seq1");
  seq2 =      xyz_sequence::type_id::create("seq2");
  seq3 =      xyz_sequence::type_id::create("seq3");

  seq1.addr_low = 'h00; seq1.addr_high = 'h60;
  seq2.addr_low = 'h20; seq2.addr_high = 'h80;
  seq3.addr_low = 'h00; seq3.addr_high = 'h80;

  phase.raise_objection(this);
  fork
    seq1.start(agent1.sequencer);
    seq2.start(agent2.sequencer);
    seq3.start(agent3.sequencer);
  join
  phase.drop_objection(this);

```

```
  endtask
endclass
```

## The top level test

The top level test is simple; register the BFM, configure anything, start the test.

```
module tests();

import uvm_pkg::*;
import axx_pkg::*;

initial begin
  axx_test::register_abc("bfm1", top.bfm1);
  axx_test::register_xyz("bfm2", top.bfm2);
  axx_test::register_xyz("bfm3", top.bfm3);
  // set_config_int("*", "recording_detail", 1);
  uvm_config_db#(int)::set(uvm_root::get(), "*", "recording_detail", 1);

  run_test("axx_test");
end
endmodule
```

## The TOP

In the top, the DUT is instantiated and connected to the BFM. The BFM are also instantiated and connected to a clock signal.

```
module top();
reg clk;

abc_if bfm1(clk); // BFM - the Bus Functional Model(s)
xyz_if bfm2(clk); // BFM
xyz_if bfm3(clk); // BFM

dut_mem dut(
  .CLK1(clk),
  .RST1(bfm1.RST),
  .RW1(bfm1.RW),
  .READY1(bfm1.READY),
  .VALID1(bfm1.VALID),
  .ADDR1(bfm1.ADDR),
  .DATA1(bfm1.DATA1),
  .DATA01(bfm1.DATA0),
  .CLK2(clk),
```

```

.RST2 (bfm2.RST),
.RW2 (bfm2.RW),
.READY2 (bfm2.READY),
.VALID2 (bfm2.VALID),
.ADDR2 (bfm2.ADDR),
.DATAI2 (bfm2.DATAI),
.DATAO2 (bfm2.DATAO),

.CLK3 (clk),
.RST3 (bfm3.RST),
.RW3 (bfm3.RW),
.READY3 (bfm3.READY),
.VALID3 (bfm3.VALID),
.ADDR3 (bfm3.ADDR),
.DATAI3 (bfm3.DATAI),
.DATAO3 (bfm3.DATAO)

); // DUT

// Turn on the clock.
always begin
#10 clk <= 1;
#10 clk <= 0;
end
endmodule

```

## Compiling and Simulating

In order to compile and simulate the ABC, XYZ and AXX verification IP must be compiled (the ABC Package, the XYZ Package and the AXX Package). Then the tests, the top and the BFM themselves are compiled.

Compile the DUT

```
vlib work
vlog ../dut/rtl.sv
```

Compile the ABC VIP, the XYZ VIP and the AXX VIP

```
vlog +incdir+../abc_pkg \
+incdir+../xyz_pkg \
..../abc_pkg/abc_pkg.sv \
..../xyz_pkg/xyz_pkg.sv \
+incdir+../axx_pkg \
..../axx_pkg/axx_pkg.sv
```

Compile the top, the tests, the abc interface and the xyz interface.

```
vlog top.sv tests.sv \
..../abc_bfm/abc_if.svh \
..../xyz_bfm/xyz_if.svh
```

Simulate the top with the tests (dual top simulation)

```
vsim -c top tests -do "run -all; quit -f"
```

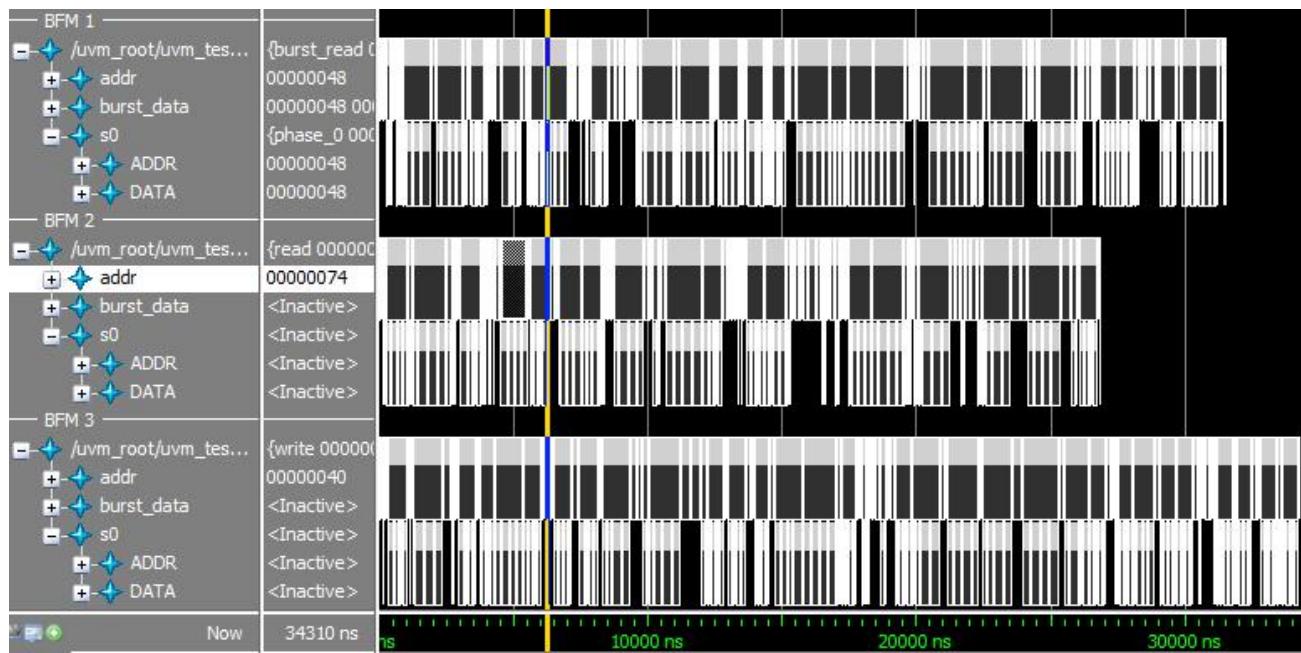
## Simulation results

The resulting simulation waveforms show randomized traffic - reads, writes, burst\_reads and burst\_writes randomly interleaved.

BFM 1 and BFM 2 are shown below.



The transaction view below shows all three BFM.



( download source code examples online at <http://verificationacademy.com/uvm-ovm> ).

# Appendix - Deployment

## OVM2UVM

A Roadmap for upgrading to UVM - this guide covers the minimum steps to upgrade your VIP and testbench from OVM to UVM compatibility, then goes into more detail on some further steps for UVM conformance.

### Migrating to UVM Chapter contents:

[OVM2UVM \(this page\)](#) - minimum steps for compatibility with UVM

[OVM2UVM/DeprecatedCode](#) - areas of OVM code that are in the process of being phased out in UVM

[OVM2UVM/SequenceLibrary](#) - complete change to the sequence library approach from OVM to UVM

[OVM2UVM/Phasing](#) - adopting basic UVM phasing

[OVM2UVM/ConvertPhaseMethods](#) - scripting to automate conversion to UVM-style phase methods

The UVM 1.0 release is intended to be backwards compatible with the functionality present in OVM 2.1.1, with the UVM 1.0 functionality being a superset of the original OVM 2.1.1 functionality. In order to run OVM code with the UVM, all of the name strings used by the OVM have to be changed from "ovm" to "uvm" to pick up their equivalent in the UVM class library - for instance the ovm\_pkg has to be changed to the uvm\_pkg and so on. With the release of OVM 2.1.2 and UVM 1.1, the same changes must be made.

A conversion script is provided in the UVM distribution to allow OVM users to translate testbench code written for OVM 2.1.1 or 2.1.2 so that it can be compiled and simulated with the UVM class library. This script does most of the translation but it does not catch everything, and the known issues are documented on this page.

Although UVM 1.1 is compatible with OVM 2.1.1, any deprecated classes or methods in OVM 2.1.1 are not present in the UVM and support for legacy classes and methods associated with the AVM or the URM have also been dropped.

Converting your OVM code to UVM is a straightforward process, but it is unidirectional. The required steps are outlined here:



- Check for deprecated features
- Check objection code
- Run OVM-to-UVM Conversion Script
- Check results
- Run Perl script to convert phasing code
- Check results
- Replace global\_stop\_request() and stop() tasks
- Update raise/drop calls to use UVM API
- Convert set/get\_config\_\*() to config\_db
- Use new UVM features

## Step 1: Audit Your OVM Code



Although UVM 1.1 is compatible with OVM 2.1.1, any deprecated classes or methods in OVM 2.1.1 or 2.1.2 are not present in the UVM. Support for legacy classes and methods associated with the AVM or the URM have also been dropped. A full list can be found in the OVM kit in the 'deprecated.txt' file. Some of the more prevalent changes are

- Removal of the `ovm_threaded_component` base class
- Removal of the `import/export_connections()` and `configure()` phase methods for `ovm_component`
- Removal of `ovm_component::find_component()` and other methods.

In general, these deprecations will only affect code that was originally developed in AVM or a pre-2.1.1 version of OVM. See also OVM2UVM/DeprecatedCode for more information.

**Summary:** The UVM makes a clean break from the OVM, any deprecated code identified in the 'deprecated.txt' file from the OVM 2.1.1 release tree will not be present in the UVM class library. If your OVM code was originally translated from the AVM, then there is a risk that you may have some AVM code which has survived, examples of which include `ovm_threaded_components`, and references to AVM phases such as `export_connections()`.

The sequencer `is_locked()` method has been deprecated and the `has_lock()` method should be used instead. Strictly speaking this is an OVM deprecation, but it is not documented in the 'deprecated.txt' file.

A copy of the `deprecated.txt` file can be downloaded as a tarball via the link below:

( **download source code examples online at** <http://verificationacademy.com/uvm-ovm> ).

The first step in converting to UVM is to audit your code and remove/change any of these deprecated features so that all of your OVM constructs have a parallel in UVM. You should rerun your simulation to ensure that everything still works as expected in OVM. Once you are satisfied with your OVM code, move on to step 2.

## Step 2: Using The `ovm2uvm` Conversion Script



The UVM distribution kit contains a perl script call `ovm2uvm.pl` which is located in the `/bin` directory. This script runs through each file with a `.sv` or `.svh` extension in the OVM test bench code and converts it to UVM by translating "ovm" to "uvm". The script has a number of options:

Option	Effect
--top_dir=<directory_name>	Converts all the files with .sv and .svh extensions in the directory tree under <directory name>
--write	Overwrites the existing files with uvm versions (the default is not to do this)
--backup	Create a backup of the ovm source files before over-writing them
--all_text_files	Also goes through text files such as scripts and make files converting ovm to uvm

The recommended way to use the script is with all these options. If --top\_dir is not specified, then the script will overwrite all the files from the current directory downwards.

```

#
# Assuming that the OVM source code is in the directory tb_build
#
$UVM_HOME/bin/ovm2uvm.pl --backup --all_text_files --write --top_dir=tb_build

# Where $UVM_HOME contains the directory path to the UVM installation

```

When the script has completed it will have done the following:

- All strings containing ovm in .sv and .svh files are converted to uvm
- Files having a name containing ovm will be changed to uvm (e.g. ovm\_container.sv will be converted to uvm\_container.sv)
- Directory names containing ovm are left unchanged
- A patch file is generated, listing the before and after differences for the conversion
- If the backup option is specified, then a backup tarball will have been created containing the original file set
- If the all\_text\_files option is specified, then any text files will have had their ovm strings changed to uvm

## Known Script Limitations

The ovm2uvm conversion script is not perfect and the following names are known to be missed by it:

OVM Name	UVM Name	Comment
analysis_fifo #(type)	uvm_tlm_analysis_fifo #(type)	
WARNING	UVM_WARNING	Reporting action enum
ERROR	UVM_ERROR	Reporting action enum
NO_ACTION	UVM_NO_ACTION	Reporting action enum (NO_ACTION is from AVM compatibility file)
DISPLAY	UVM_DISPLAY	Reporting action enum (DISPLAY is from AVM compatibility file)
LOG	UVM_LOG	Reporting action enum (LOG is from AVM compatibility file)
COUNT	UVM_COUNT	Reporting action enum (COUNT is from AVM compatibility file)
EXIT	UVM_EXIT	Reporting action enum (EXIT is from AVM compatibility file)
CALL_HOOK	UVM_CALL_HOOK	Reporting action enum (CALL_HOOK is from AVM compatibility file)

This list will be updated if more escaping names are discovered.

## Script Side-Effects

The script produces some side effects in the converted files which you will have to work through. Examples of these side effects include:

- If a `include statement includes a path that spans a directory path with 'ovm' somewhere in the string, then the path will no longer be valid since the directory name would not have changed
- The raise\_objection() and drop\_objection() methods in UVM have a different signature than in OVM. If you are using objections in your OVM code, you will have to modify the calls (including raised(), dropped() and all\_dropped()) manually to account for the change. See here for a list of API differences.
- If you have deprecated methods that can be migrated, then the function name will be changed but if the 'endfunction: <method\_name>' coding style is used the end label string will not be updated

Scripts and make files that are updated because of the all\_text\_files option should be checked to ensure that, for instance:

- Directory paths containing ovm have not been changed to uvm
- Paths to what was OVM package source code are now pointing to the UVM package source code
- Questa OVMdebug switches will be converted to UVMdebug - this will only work in versions of the simulator that support UVM\_DEBUG

## Other Migration Steps

Although running the conversion script migrates the testbench code, there are other changes to the overall compile and simulation flow:

- Environment variables such as OVM\_HOME need to be replaced with UVM\_HOME
- The UVM package source will have to be compiled (at least in the short term)
- The simulation command line switch to choose the test class changes from +OVM\_TESTNAME to +UVM\_TESTNAME
- The UVM uses a shared object to support some of the new features such as the PLI code to do backdoor register accesses. As of Questa 10.0c, this shared object will be loaded automatically. In earlier Questa versions, the shared object needs to be specified at the simulator command line. In Questa this is done using the -sv\_lib switch which, by default, should point to the location of the UVM dpi shared object (\$UVM\_HOME/lib/uvm\_dpi)
- If run\_test() is called after time 0, that will cause an error in UVM. The top level module code which calls run\_test() will need to be updated to call it at time 0. The UVM testbench code may also need to be updated to synchronize with the previous simulation time when run\_test() was called an OVM testbench.
- If your OVM code does not use objections (i.e., you use global\_stop\_request() to stop the simulation), you must add the +UVM\_USE\_OVM\_RUN\_SEMANTIC command-line argument. If you use objections in your OVM code, you'll need to proceed through step 4.

## +UVM\_USE\_OVM\_RUN\_SEMANTIC Plusarg

In OVM, there were two user initiated ways to stop a testbench when the time comes. The first and oldest way is to call the ovm\_pkg::global\_stop\_request() method. This method will cause all run() tasks to be terminated. The other way which was introduced in OVM 2.1 is to use objections, specifically the ovm\_test\_done objection. If the testbench uses global\_stop\_request(), then after the conversion the testbench will not run correctly. This is because by default the only accepted user initiated way to stop a test in UVM is to use objections. To enable the use of the global\_stop\_request(), +UVM\_USE\_OVM\_RUN\_SEMANTIC has to be added to the simulation command line. If ovm\_test\_done was used, this will be converted to uvm\_test\_done by the script. The testbench will work, but is not fully converted to the new

UVM phasing scheme.

### Creating the UVM Shared object file

**NOTE:** This is only necessary when using a Questa version 10.0b or earlier.

The shared object file will need to be created for the simulator version and the computing platform in use. The UVM installation package has a set of simulator specific make files in the \$UVM\_HOME/examples directory and these will compile the object and place it in the \$UVM\_HOME/lib directory. The name of resultant file will be uvm\_dpi.so in the case of a linux shared object and the make file will generate the appropriate extension for other operating systems.

```
#  
# To compile the UVM shared object file for Questa  
#  
cd $UVM_HOME/examples  
make -f Makefile.questa dpi_lib  
  
#  
# This will create a file called $UVM_HOME/lib/uvm_dpi.so (on a Linux platform)  
#
```

### Running a UVM simulation with Questa:

As an example - In order to run a Questa simulation the following steps will need to be taken:

```
#  
# $UVM_HOME = <path_to_uvm_install>  
#  
# Compile the UVM package:  
vlog +incdir+$UVM_HOME/src $UVM_HOME/src/uvm_pkg.sv  
  
# Compile the rest of the test bench code ...  
  
# Simulate the converted test bench:  
vsim my_ovm_tb +UVM_TESTNAME=<my_test>
```

## Step 3: Convert to UVM Phasing API



UVM modifies the phase method calls from OVM in two important ways:

1. The method name is changed to "<name>\_phase"
2. An argument is added to all phase methods

**OVM phase methods:**

```
class my_component extends ovm_component
  `ovm_component_utils(my_component)
  ...
  extern function void build();
  extern function void connect();
  extern task run;
endclass

function void my_component::build();
  super.build();
  ...
endfunction

...
```

**UVM phase methods:**

```
class my_component extends uvm_component
  `uvm_component_utils(my_component)
  ...
  extern function void build_phase(uvm_phase phase);
  extern function void connect_phase(uvm_phase phase);
  extern task run_phase(uvm_phase phase);
endclass

function void my_component::build_phase(uvm_phase phase);
  super.build_phase(phase);
  ...
endfunction

...
```

The necessary changes can be made to your OVM code by running the following perl script:

```
#!/usr/bin/perl
#
# This perl script converts OVM to UVM phase method names and super.phase() calls
# on one SV source file. By default it reads input, translated, and writes output.
#
# To invoke it for 'in-place conversion' of a file or files, do the following:
# A backup of your sourcefile(s) will be saved in FILENAME.bak
# Convert one file in-place:
#   % perl -i.bak this_script.pl your_sourcefile.sv
# Convert a set of files:
#   % perl -i.bak this_script.pl *.sv*
# Convert a whole tree of files:
#   % find . -name "*.sv*" -exec perl -i.bak this_script.pl {} \;
#
# NOTE: both the method declaration AND the super calls are converted - if you
# do the one you must do the other too, no mixing and matching.
#
$PHASES='build|connect|end_of_elaboration|start_of_simulation|extract|check|report';
while(<>) {
  #convert phase function declarations
  s/(\Wfunction\s+.*($PHASES)) (\s*\(\)/${1}_phase${3}uvm_phase phase/;
  # convert run phase task declaration
  s/(\Wtask\s+.*(run)) (\s*\(\)/${1}_phase${3}uvm_phase phase/;
  # convert super.phase calls
  s/(super.($PHASES)) (\s*\(\)/${1}_phase${3}phase/;
  print;
}
```

## Script Improvements to be made

As the script is run on more code, it can be enhanced to cover unanticipated scenarios:

- convert endfunction/endtask tags (e.g. endfunction : connect)
- ensure user methods called xxxxPHASE are not converted

## Step 4: Use Objections



UVM 1.1 uses objections to control the ending of task-based phases. Step 6 will discuss adding the new UVM task-based phases to your code. If you are porting OVM code to UVM, you are currently either using the `ovm_test_done` objection (see `Ovm/EndOfTest` for the recommended solution) or `global_stop_request()` to control the end of the run phase.

### Use `phase_ready_to_end()` instead of `stop()`

In the OVM, passive components not involved in the raise/drop objection game would use the `stop` method to delay End-Of-Test. For example, we might do something like this:

```
task run();
  ...
  busy = 1;
  ...
  busy = 0;
  ...
endtask

task stop();
  wait( busy == 0 );
endtask
```

In the UVM, we cannot have a single `stop` method since we now have multiple phases. The code above will work in the UVM but will only work for the `run` phase. In the UVM, we have two functions `phase_ready_to_end` and `phase_ended`, both of which take a `uvm_phase` as their only argument. Passive ( in the sense that they do not raise and drop objections during the ordinary lifetime of time consuming phases ) components should use the `phase_ready_to_end` and `phase_ended` methods to delay the end of a phase. Refer to the Phasing/Transactors page for the best way to do this.

## Do not use `global_stop_request`

In the OVM, calling `global_stop_request` was not recommended for large testbenches although it was not deprecated. In the UVM, it works but is now deprecated since it really is not consistent with having multiple time consuming phases.

A simple testbench in the OVM might use `global_stop_request` in `run()`. With "normal" compile and elaboration options, this code will **not** work in the UVM. As an interim solution, you need to add `+UVM_USE_OVM_RUN_SEMANTICS` to the command line. The recommended UVM style, which does not require any extra command line options, is to use objections:

### OVM `global_stop_request`:

```
task run();
    seq.start( m_virtual_sequencer );
    global_stop_request();
endtask
```

### UVM run phase objection:

```
task run_phase( uvm_phase phase );
    phase.raise_objection( this );
    seq.start( m_virtual_sequencer );
    phase.lower_objection( this );
endtask
```

A longer discussion of End-of-test is available at [EndOfTest](#).

## Do not use `uvm_test_done`

The `uvm_test_done` objection works in the UVM, but it is not the recommended way of doing things. In the UVM there are many time consuming phases, so using a global variable is no longer a robust mechanism. The recommended way to do things in the UVM is to use the phase-specific objection to control the run phase. For example in the OVM using `uvm_test_done` would be converted to UVM as:

### OVM test done objection:

```
task run();
    uvm_test_done.raise_objection( this );
    seq.start( m_virtual_sequencer );
    uvm_test_done.drop_objection( this );
endtask
```

### UVM phase objection:

```
task run_phase( uvm_phase phase );
    phase.raise_objection( this , "started sequence" );
    seq.start( m_virtual_sequencer );
    phase.drop_objection( this , "finished sequence" );
endtask
```

## Objection differences

The uvm\_objection callback methods have an additional description argument to their ovm\_objection equivalents:

### OVM objection methods:

```
//  
// Dropped:  
function void dropped(  
    ovm_objection objection,  
    uvm_object source_obj,  
    int count);  
  
// Raised:  
function void raised(  
    uvm_object obj,  
    uvm_object source_obj,  
    int count);  
  
// Raise objection:  
function void raise_objection(  
    uvm_object obj = null,  
    int count = 1);  
  
// Drop objection:  
function void drop_objection(  
    uvm_object obj=null,  
    int count = 1);  
//
```

### UVM objection methods:

```
//  
// Dropped:  
function void dropped(  
    uvm_objection objection,  
    uvm_object source_obj,  
    string description,  
    int count);  
  
// Raised:  
function void raised(  
    uvm_object obj,  
    uvm_object source_obj,  
    string description,  
    int count);  
  
// Raise objection:  
function void raise_objection(  
    uvm_object obj = null,  
    string description = "",  
    int count = 1);  
  
// Drop objection:  
function void drop_objection(  
    uvm_object obj=null,  
    string description = "",  
    int count = 1);
```

The new second argument in the raise\_objection() and drop\_objection() methods provides advantages for debug.

## Step 5: Use uvm\_config\_db



OVM used the [set,get]\_config\_[int,string,object] methods for configuring components. The UVM equivalents of these methods are available, but not recommended. Instead, UVM supports the uvm\_config\_db database API, which supplies the same functionality with several key advantages.

The uvm\_config\_db is parameterized by the type of object that is being configured. On get() calls, that means that the return-type is specified in the call and thus so \$cast is required.

**OVM set/get config:**

```

class my_env extends ovm_env;
...
function void build();
    ahb_cfg = ahb_config::type_id::create("ahb_cfg");
    ahb_cfg.width = 16;
    // set additional fields
    set_config_object("*", "ahb_cfg", ahb_cfg);
endfunction
...
endclass

class my_ahb_agent extends ovm_component;
...
function void build();
    ovm_object cfg;
    ahb_config my_cfg;
    assert(get_config_object("ahb_cfg", cfg, 0));
    if (!$cast(my_cfg, cfg))
        ovm_report_error(...);
...
endfunction
...
endclass

```

**UVM config DB:**

```

class my_env extends uvm_env;
...
function void build();
    ahb_cfg = ahb_config::type_id::create("ahb_cfg");
    ahb_cfg.width = 16;
    // set additional fields
    uvm_config_db#(ahb_config)::set(
        this, "ahb_agent", "ahb_cfg", ahb_cfg);
endfunction
...
endclass

class my_ahb_agent extends uvm_component;
...
function void build();
    ahb_config my_cfg;
    if (!uvm_config_db::ahb_config::get(
        this, "", "ahb_cfg", my_cfg));
        `uvm_error(...);
...
endfunction
...
endclass

```

For more information on using `uvm_config_db`, see the the page [Resources/config db](#).

## Step 6: Add Additional UVM Functionality



UVM adds several additional features beyond what is available in OVM. The final step in migrating your OVM code to UVM is to add some of these new UVM-only features.

### UVM Messaging Changes

To improve performance, the `uvm_report_[info, warning, error, fatal]` methods have been replaced by macros with the same arguments. The macros provide two explicit advantages.

1. The macro can determine if the verbosity would allow the message to be printed before processing the text string, which is often a `$sformat` or similar call.
2. The macro allows the message handler to include the file and line number of the call automatically.

Method-based message code such as

```
uvm_report_info("PKT", $sformat("Packet[%d] Sent", i), UVM_HIGH);
```

would be replaced by

```
`uvm_info("PKT", $sformat("Packet[%d] Sent", i), UVM_HIGH)
```

and produce the following output:

```
UVM_INFO myfile.sv(15) @10 uvm_test_top.test.generator [PKT]: Packet[1] Sent
```

## Sequence Library Differences

The old OVM sequence library has been completely deprecated, and is not recommended for new designs. A new `uvm_sequence_library` is in development in UVM, for more details see OVM2UVM/SequenceLibrary.

## Command Line Processor

UVM 1.1 provides a utility for handling command-line plusargs directly. Please see UVM/CommandLineProcessor for more details.

## Registers

The next feature you may wish to add to your migrated verification code is the new UVM register layer. This feature is described in detail Registers.

# OVM2UVM/DeprecatedCode

---

This page is part of a group of pages describing the overall OVM-to-UVM Migration flow.

Accellera UVM1.0 used OVM2.1.1 as its basis, with the intention of preserving backwards compatibility where possible. Some areas have broken backwards compatibility, in order to achieve better forwards-compatibility. A policy of deprecation has been used in several places to make this migration as easy as possible while steering the user community towards future native UVM methodology.

There are two kinds of deprecation that are worthy of documentation: deprecation of current UVM code features, and removal of previous OVM/AVM deprecated features.

## Deprecation of Current UVM code

There are a number of sections of code in the UVM1.0 release that are marked as deprecated.

### What does deprecation mean

- Code/classes/APIs which do not form part of the UVM standard
- They exist only to ease migration from OVM2.1.1
- They may be removed altogether in **any** future UVM release as determined by Accellera committee.
- Therefore, this code should not be recommended for use in new designs

### What UVM features are deprecated

- OVM-style sequence library - see OVM2UVM/SequenceLibrary

### How can I test out my code for future compatibility

You can add a `+define+UVM_NO_DEPRECATED` to a vlog compile of the UVM library (this means you will have to download your own UVM library rather than use the precompiled one bundled with Questa).

## Previous OVM/AVM deprecated features

- OVM code that was already marked as deprecated was removed when UVM development was started.
- Older OVM testbenches or those from AVM legacy may use some of these features and need manual or scripted migration to new base-classes / APIs and in some cases more complex rewrites of sections of code.

# OVM2UVM/SequenceLibrary

---

Updating your VIP/testbench sequence library is one task that you may have to perform while migrating from OVM to UVM. This page should give you all the information you need - it is part of a group of pages describing the overall OVM-to-UVM migration

## The OVM Sequence Library is deprecated

The OVM Sequence Library has been deprecated in UVM - it is not part of the UVM standard and never will be, although the old code is still present in the UVM base class library.

### What exactly is deprecated

The following macros/classes/APIS are deprecated and therefore should not be used in new designs:

The OVM sequence library macros:

- `uvm\_sequencer\_utils / `uvm\_sequencer\_param\_utils
  - (if you really need to extend uvm\_sequencer (why?), use `uvm\_component\_utils)
- `uvm\_sequence\_utils
  - (use `uvm\_object\_utils instead)
- `uvm\_declare\_sequence\_lib / `uvm\_update\_sequence\_lib / `uvm\_sequence\_lib\_and\_item
  - (use the uvm\_sequence\_library)

The OVM builtin sequences:

- class uvm\_random\_sequence
- class uvm\_exhaustive\_sequence
- class uvm\_simple\_sequence

The OVM sequence-library-supporting sequence API:

- uvm\_sequence\_base::seq\_kind
- uvm\_sequence\_base::num\_sequences()
- uvm\_sequence\_base::get\_seq\_kind()
- uvm\_sequence\_base::get\_sequence()
- uvm\_sequence\_base::do\_sequence\_kind()
- uvm\_sequence\_base::get\_sequence\_by\_name()

The entire OVM sequencer 'count' and 'default\_sequence' config mechanism:

- uvm\_sequencer\_base::count
- uvm\_sequencer\_base::max\_random\_count
- uvm\_sequencer\_base::max\_random\_depth
- uvm\_sequencer\_base::default\_sequence
- uvm\_sequencer\_base::assorted APIs

## Why was this done

This action was taken in Accellera to facilitate replacing the old, limited capability with a redesigned capability for forward compatibility.

## The new UVM sequence library

The replacement is a class (uvm\_sequence\_library) which is a sequence and hence is more versatile.

Refer to src/seq/uvm\_sequence\_library.svh for details - there are many comments indicating the use model. The introduction (from uvm1.0 release) is included here:

```
// CLASS- uvm_sequence_library
//
// The ~uvm_sequence_library~ is a sequence that contains a list of registered
// sequence types. It can be configured to create and execute these sequences
// any number of times using one of several modes of operation, including a
// user-defined mode.
//
// When started (as any other sequence), the sequence library will randomly
// select and execute a sequence from its ~sequences~ queue. If in
// <UVM_SEQ_LIB_RAND> mode, its <select_rand> property is randomized and used
// as an index into ~sequences~. When in <UVM_SEQ_LIB_RANC> mode, the
// <select_randc> property is used. When in <UVM_SEQ_LIB_ITEM> mode, only
// sequence items of the ~REQ~ type are generated and executed--no sequences
// are executed. Finally, when in <UVM_SEQ_LIB_USER> mode, the
// <select_sequence> method is called to obtain the index for selecting the
// next sequence to start. Users can override this method in subtypes to
// implement custom selection algorithms.
//
// Creating a subtype of a sequence library requires invocation of the
// <`uvm_sequence_library_utils> macro in its declaration and calling
// the <init_sequence_library> method in its constructor. The macro
// and function are needed to populate the sequence library with any
// sequences that were statically registered with it or any of its base
// classes.
//
// class my_seq_lib extends uvm_sequence_library #(my_item);
// `uvm_object_utils(my_item)
// `uvm_sequence_library_utils(my_seq_lib)
// function new(string name="";
// super.new(name);
// init_sequence_library();
// endfunction
// ...
// endclass
```

# OVM2UVM/Phasing

---

## Porting OVM code to Phase Aware UVM code

OVM code can be ported to run on the UVM. With some exceptions, this is a matter of avoiding features ( such as the AVM and uRM backward compatibility layers ) that were deprecated in the OVM and then converting all instances of the ovm prefix to uvm. This article describes a further stage in the porting process, where the resulting OVM code is gradually converted to make it more UVM compliant. Each step results in a working UVM testbench, so we would expect that after each stage regressions are rerun to ensure that the testbench is still functioning correctly. The end result of this process is that we end up with a UVM testbench as described in the UVM phasing page.

## Basic Porting

If we follow the simple 'minimum porting' as described in OVM2UVM, we will end up with code that works in the UVM but which uses features of the UVM which are deprecated, such as `global_stop_request`, the `stop` method and the `uvm_test_done` objection. This section explains how to change this code so that it no longer uses these deprecated features of the UVM.

## Introduce Phase Arguments

In the OVM, phase methods ( eg `build`, `connect`, `run` etc ) had no arguments. In the UVM, the phase methods have the suffix `_phase` and have a `uvm_phase` argument. In time consuming phases, this argument is used to raise and lower objections to the phase ending.

For example, OVM code that looks like this:

```
class my_component extends ovm_component;
  `ovm_component_utils( my_component )
  ...
  extern function new( string name , ovm_component parent = null );
  extern function void build();
  extern function void connect();
  extern task run();
endclass
```

Will be translated to look like this:

```
class my_component extends uvm_component;
  `uvm_component_utils( my_component )
  ...
  extern function new( string name , uvm_component parent = null );
  extern function void build();
  extern function void connect();
  extern task run();
endclass
```

To avoid the deprecated form of the phase methods, this code should be hand-edited to look like this:

```
class my_component extends uvm_component;
  `uvm_component_utils( my_component )
  ...
  extern function new( string name , uvm_component parent = null );
  extern function void build_phase( uvm_phase phase );
  extern function void connect_phase( uvm_phase phase );
  extern task run_phase( uvm_phase phase );
endclass
```

**Note:** A script to convert the phase methods can be found on the OVM2UVM/ConvertPhaseMethods page.

## Use Phase Objections not ovm\_test\_done

The recommended OVM End-of-Test mechanism uses `ovm_test_done.raise_objection`, `ovm_test_done.drop_objection` and the `stop` method.

The `ovm_test_done` mechanism works in the UVM, but it is not the recommended way of doing things. In the UVM there are many time consuming phases, so using a global variable is no longer a robust mechanism. The recommended way to do things in the UVM is to use the phase argument to the `run` phase. For example in the OVM we may have written:

```
task run();
  ovm_test_done.raise_objection( this );
  seq.start( m_virtual_sequencer );
  ovm_test_done.drop_objection( this );
endtask
```

In the UVM, we would recommend:

```
task run_phase( uvm_phase phase );
  phase.raise_objection( this , "started sequence" );
  seq.start( m_virtual_sequencer );
  phase.drop_objection( this , "finished sequence" );
endtask
```

[ Note : we can take advantage of the new second argument in the `raise_objection` and `drop_objection` methods to help with debug ]

## Use phase\_ready\_to\_end() instead of stop()

In the OVM, passive components not involved in the `raise/drop` objection game would use the `stop` method to delay End-Of-Test. For example, we might do something like this:

```
task run();
  ...
  busy = 1;
  ...
endtask
```

```

busy = 0;
...
endtask

task stop();
  wait( busy == 0 );
endtask

```

In the UVM, we cannot have a single stop method since we now have multiple phases. The code above will work in the UVM but will only work for the run phase. In the UVM, we have two functions `phase_ready_to_end` and `phase_ended`, both of which take a `uvm_phase` as their only argument. Passive ( in the sense that they do not raise and drop objections during the ordinary lifetime of time consuming phases ) components should use the `phase_ready_to_end` and `phase_ended` methods to delay the end of a phase. The best way to do this is documented here.

## Do not use `global_stop_request`

In the OVM, calling `global_stop_request` was not recommended for large testbenches although it was not deprecated. In the UVM, it works but is now deprecated since it really is not consistent with having multiple time consuming phases.

A simple testbench in the OVM might have done this:

```

task run();
  seq.start( m_virtual_sequencer );
  global_stop_request();
endtask

```

With "normal" compile and elaboration options, this code will **not** work in the UVM. To get it to work, you need to add `+UVM_USE_OVM_RUN_SEMANTICS` to the command line.

The recommended UVM style, which does not require any extra command line options, is to use objections:

```

task run_phase( uvm_phase phase );
  phase.raise_objection( this );
  seq.start( m_virtual_sequencer );
  phase.drop_objection( this );
endtask

```

A longer discussion of End-of-test is available here.

## Become Phase Aware

### Introduce Multiple Phases Into Test

One of the most significant differences between the OVM and the UVM is that the UVM has multiple time consuming phases. The main time consuming phases are reset, configure, main and shutdown. In general, transactors should continue to execute in the `run_phase`, but sequences should execute in one of those four main time consuming phases. A first pass refactoring when we port code from OVM to UVM might be to move the execution of the test from the `run_phase` to the `main_phase`. Further refactoring might result in this code being distributed between multiple phases. So code

which started off like this in the OVM:

```
task run();
    reset_seq.start( m_sequencer );
    program_control_registers_seq.start( m_sequencer );
    data_transfer_seq.start( m_sequencer );
    read_status_registers.start( m_sequencer );
    global_stop_request();
endtask
```

Might now look like this:

```
task reset_phase( uvm_phase phase);
    phase.raise_objection( this );
    reset_seq.start( m_sequencer );
    phase.drop_objection( this );
endtask

task configure_phase( uvm_phase phase);
    phase.raise_objection( this );
    program_control_registers_seq.start( m_sequencer );
    phase.drop_objection( this );
endtask

task main_phase( uvm_phase phase);
    phase.raise_objection( this );
    data_transfer_seq.start( m_sequencer );
    phase.drop_objection( this );
endtask

task shutdown_phase( uvm_phase phase);
    phase.raise_objection( this );
    read_status_registers_seq.start( m_sequencer );
    phase.drop_objection( this );
endtask
```

## Review Scoreboards and Other Analysis Components

Having made the test phase-aware, there might also be some benefit in making analysis components, such as scoreboards, phase-aware as well. A discussion of this topic is here.

## Advanced

There are other advanced phasing technologies in the UVM such as phase domains, phase jumping and multiple domains. These technologies are new in the UVM and their use models are not yet well defined. Unless you are an advanced user we do not recommend that you use these features until the methodology associated with them becomes

more stable. We do not believe that these techniques will be needed when doing an OVM to UVM migration.

# OVM2UVM/ConvertPhaseMethods

Part of the OVM to UVM conversion process is to change the method names for OVM phase methods (build, connect, run, etc) to the new UVM signature for phase methods. UVM continues to support the old signature but it will ultimately be deprecated, and you may need to move to the new signature to use some UVM phasing features.

## Specification:

Converting phase methods from OVM to UVM involves the following:

1. Alter phase method declarations, adding '\_phase' suffix to name and adding 'uvm\_phase phase' parameter to signature. Convert PHASE() to PHASE\_phase(uvm\_phase phase).

1a. The following are the basic prototypes for the methods to be updated. PHASE can be any one of 8 legacy OVM phase names. No need to update the new UVM phases.

- function void build()
- function void connect()
- function void end\_of\_elaboration()
- function void start\_of\_simulation()
- function void extract()
- function void check()
- function void report()
- task run()

1b. Variations on the above need to be accommodated, so use a relaxed regular expression rather than guess them all:

- function void PHASE()
- extern function void PHASE()
- function void my\_class\_name::PHASE()
- addition of 'automatic' or other qualifiers
- addition of whitespace or /\*comments\*/ anywhere on line

1c. Some variations cannot be so easily accommodated but are less likely to occur in sensible code.

- declaration split across multiple lines are harder to match.
- multiple declarations on one line
- use of customer-side macros rather than plain typing

1d. Avoid false matches by tightening up regular expression where possible

2. As well as phase method declarations, we also need to convert super.PHASE() calls to be super.PHASE\_phase(phase)

2a. again do this only for the list of 8 legacy OVM phase names in 1a above

## Implementation: A perl script operating on one file, using in-place editing and regular expressions.

```
#!/usr/bin/perl
#
# This perl script converts OVM to UVM phase method names and super.phase() calls
# on one SV source file. By default it reads input, translated, and writes output.
#
```

```

# To invoke it for 'in-place conversion' of a file or files, do the following:
# A backup of your sourcefile(s) will be saved in FILENAME.bak
# Convert one file in-place:
#   % perl -i.bak this_script.pl your_sourcefile.sv
# Convert a set of files:
#   % perl -i.bak this_script.pl *.sv*
# Convert a whole tree of files:
#   % find . -name "*.sv*" -exec perl -i.bak this_script.pl {} \;
#
# NOTE: both the method declaration AND the super calls are converted - if you
# do the one you must do the other too, no mixing and matching.
#
$PHASES='build|connect|end_of_elaboration|start_of_simulation|extract|check|report';
while(<>) {
    #convert phase function declarations
    s/(\Wfunction\s+.*($PHASES)) (\s*\()/$1_phase${3}uvm_phase phase/;
    # convert run phase task declaration
    s/(\Wtask\s+.*(run)) (\s*\()/$1_phase${3}uvm_phase phase/;
    # convert super.phase calls
    s/(super.($PHASES)) (\s*\()/$1_phase${3}phase/;
    print;
}

```

## Improvements to be made

As the script is run on more code, it can be enhanced to cover unanticipated scenarios:

- convert endfunction/endtask tags (e.g. endfunction : connect)
- ensure user methods called xxxxPHASE are not converted

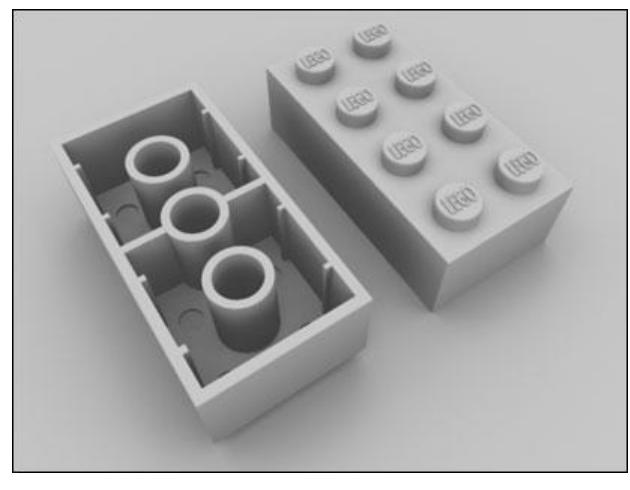
# UVC/UvmVerificationComponent

## UVM Verification Component

A UVC (UVM Verification Component) is a Verification Component designed for use in UVM. It is a multi-faceted definition and has different layers of meaning in different contexts. We define these below. The general tone of our definition is to provide malleable guidelines for reuse and a useful starting point for development, rather than rigid, innovation-stifling compliance.

### Types of UVC

There are more than one possible topologies for what we define to be a UVC, depending on the protocol and use model. These should normally boil down to the following:



#### Protocol UVC

Each verification component connects to a single DUT interface and speaks a single protocol, optionally either as a master or a slave endpoint.

#### Fabric UVC

AKA Compound UVC, a Fabric UVC is a verification component that contains a configurable number of instances of the above Protocol UVC, either in master or slave or passive mode, and configured and hooked up coherently as a unit. The purpose here is to verify a structured fabric with multiple interfaces of the same protocol.

#### Layered UVC

Provides a higher level of abstraction than the basic pin protocol. There are two common variants of construction:

- a UVC which does not connect to pins but provides an upper layer of abstraction external to an existing Protocol UVC for the lower layer.
- alternatively, a UVC which wraps and extends (by inheritance or composition) a lower-level Protocol UVC

## UVC detailed definition

Covering each aspect of the typical requirements of a UVC. These are malleable guidelines rather than strict rules.

## Relevant to native verification of the protocol

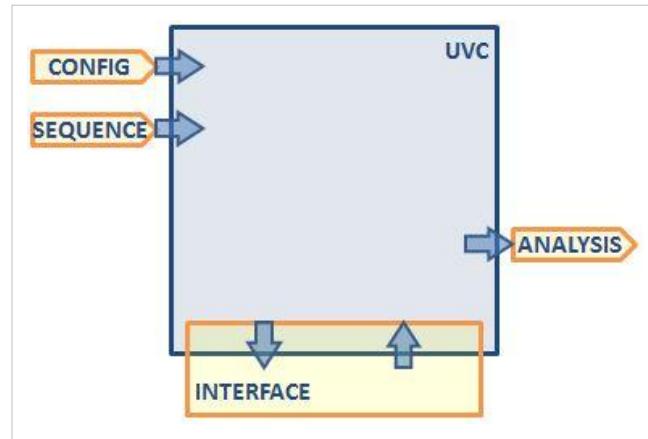
A UVC should be packaged with verification collateral and capabilities relevant to the protocol / functional domain in question. This should include, but not be limited to, the following kinds of collateral:

- A protocol-specific DUT interface. This is an SV interface construct which encapsulates all the pin-related connectivity of the UVC and enables easiest DUT connection. It may be parameterized if the DUT/protocol is configurable. It may incorporate some BFM methods if the UVC is structured for use in accelerated environments e.g. emulation.
- A Test Plan and associated compliance sequences (complex sequences which get 100% coverage according to the testplan) which may be reused vertically.
- Additional models such as slaves, memory models, arbiters, relevant to the protocol, basically whatever is necessary to show how to use the features in a fabric as per protocol specification, and to achieve 100% coverage.
- Assertions to check protocol compliance. These can be placed in the interface using SVA, or in the monitor using procedural SV code.
- Configuration settings relevant to the protocol

## Familiar to teams with UVM experience

Once the primary concerns of relevance to the verification task in hand, and applicability to the protocol or functionality, are addressed, all remaining aspects of the look and feel of the UVC at both the file manifest level, the SV language packaging, and the UVM class architecture, should be a matter of preserving familiarity. Every UVC should have the same high-level features and familiar terminology: the analysis port, the configuration class, the UVM-style sequence library, the virtual interface.

Some protocols will demand architectural departures from the familiar norm; some underlying technology may deviate from it by necessity of the business model relying on protected code, or special non-SV implementations, but familiarity from the user's point of view should be preserved if possible, often by wrapping the inner architecture in a regular agent and having configuration options to expose the optional details.



## Conformant to de facto Methodology standards

Over time the UVM methodology will develop into a set of guidelines for best practice. The Base Class Library of UVM deliberately avoids defining best practice, as this would lead to a stifling of innovation around a rigid standard. However, methodology will emerge in the form of well established patterns. Examples of these might be:

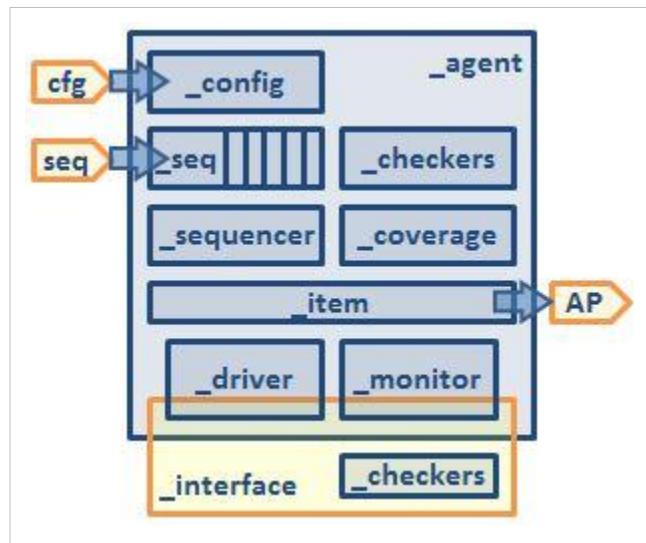
- how to hook up the DUT connection
- specifying the mode of a UVC: active (i.e. can drive the pins) or passive (i.e. will never drive the pins, only monitor)
- specify major configuration options by terminology (e.g. Master vs Slave). Such terminology is usually protocol-specific and so is not encoded rigidly in the UVM base classes.

## Consistent with UVM methodology intent

The architecture of a Protocol UVC is intended to include the following elements to match the requirements of behavior:

- a driver component, for translating abstract transactions retrieved from the sequencer, into pin-level protocol,
- a monitor component, for reassembling and reporting abstract transactions from observed pin-level protocol onto an analysis port,
- an interface, incorporating logical pin connections and optional bfm methods to interact with them
- a data item base class, capturing the abstract transaction, random constraints, behavioral aspects common to driver and monitor
- a uvm\_sequencer #(item) instance, to feed the driver with designated ready sequence traffic
- a library of base 'API' sequences for intrinsic behavior to use as building blocks in more complex stimulus
- a configuration class covering protocol-specific and test environment control knobs, shared throughout the agent, and containing the virtual interface if appropriate.
- an optional coverage object to collect, trigger and sample protocol transaction coverage
- an optional single-ended checker if the protocol demands it, reporting higher level protocol fails.
- all of these components optionally wrapped in an agent component
- Everything `included into a package which is part of an organized package hierarchy.

Refer to Agent for more information on these structures.

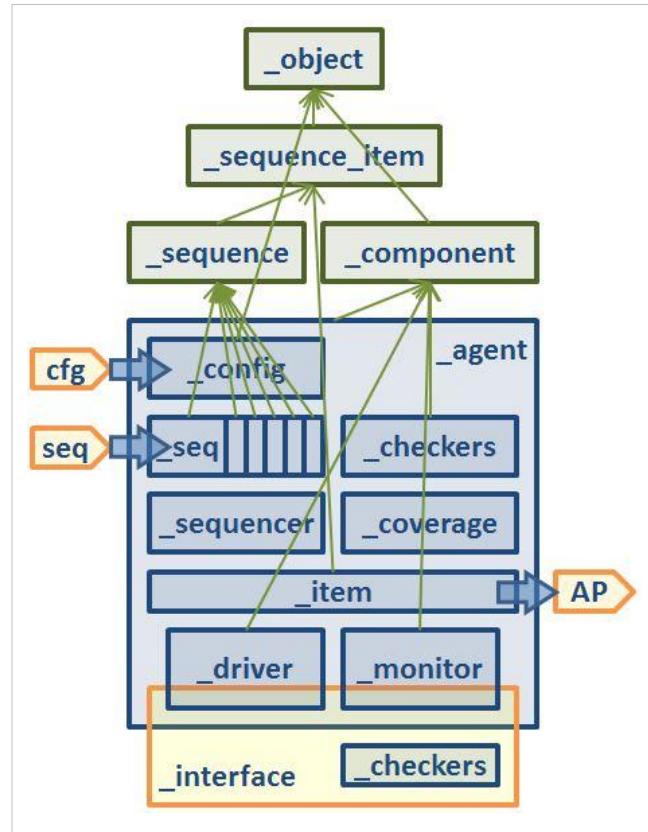


## Compatible with UVM Base Class API

A UVC is more than just one `uvm_component`. A UVC consists of several classes which extend the UVM Base Class Library base classes for structural components, data objects, and other types. Note that it is somewhat arbitrary to the definition of a UVC, whether the user extends the base `uvm_component` class, or uses the vacuous base classes for `uvm_monitor`, `uvm_driver`, `uvm_agent`. There is some value in using those classes as (a) users and tools can recognize the intent of the code and assist with development and debug, and (b) future capabilities for configuration and control can be built into the UVM library and UVCs will all benefit from those behaviors.

A UVC makes use of the factory registration API and macros provided with UVM so that its components and data can participate in factory creation. This always constrained by the particular UVC architecture. Ultimately the user can decide on what portions may be factory substituted and what may not.

A UVC uses standard TLM and Analysis port connections where required, rather than a bespoke connection technique. All protocols will have at least one analysis port. Some protocols may have more than one, either because of (a) phases within the protocol (e.g. a split address/data phase) or (b) layers within the protocol (e.g. a physical layer, transport layer, and message layer).



## What's NOT in a Protocol UVC?

### Outer Environment Layers

Simple Protocol UVCs are normally agents, not environments. They are instantiated singly, one per interface.

The environment they should be instantiated within is the user's test environment for that DUT testbench. Only the user can know what should be in that environment. Only the user can know what sub-environments may be defined to allow vertical reuse of a collection of agents and related analysis components.

By contrast, Fabric UVCs are a collection of simple Protocol UVCs (agents) pre-configured within an environment layer, designed to connect to all ports of a well-defined DUT representing a topological fabric of homogeneous protocol endpoints (in master or slave mode). That configurable environment may be a deliverable with the Protocol UVC. However, normal usage of that Protocol UVC for DUTs other than a standard bus fabric, would use the simple agent, not the environment.

## Scoreboards

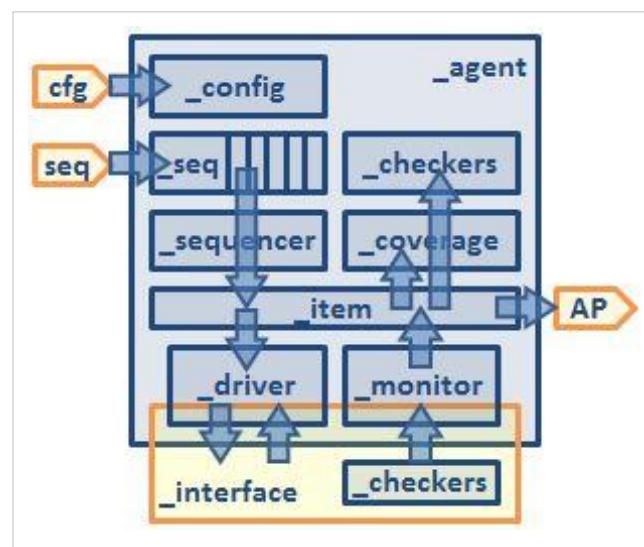
UVCs normally do not contain a scoreboard - by definition a scoreboard is a checker which compares two or more streams of monitored activity for expected cause and effect.

Some use the term 'scoreboard' to cover single-ended checkers - UVCs can contain such checkers but only for protocol checks (normally in the interface assertions or in procedural code in the monitor) and/or self-consistency checks (e.g. for register/memory model VCs, or for normalization or checksumming that spans multiple reported transactions)

By contrast, Fabric UVCs which verify a fabric of homogeneous interfaces (with master transactions in and slave transactions out) may **supply** a scoreboard and optionally instantiate it in the environment to suit the configured fabric topology. [in this case the two or more streams of monitored activity are from the same UVC type, so in this instance the addition of a scoreboard makes the Fabric UVC more of a complete verification solution than it would otherwise be]. The scoreboard is not instantiated within a single UVC agent, but in the enclosing environment object (Fabric UVC).

## Example Protocol UVC Code

The following code represents an example of a simple UVC. Non-relevant code has been removed for clarity (e.g. uvm macros, constructors). For more information on organization of UVC packages, visit the Package/Organization article.



```
//my_uvc_pkg.sv

`include "my_interface.svh"
//Include the sequence_items (transactions)
`include "my_data.svh"

package my_uvc_pkg;
  //Include the agent config object
  `include "my_config.svh"
  //Include the API sequences
  `include "my_sequence.svh"
  //Include the components
  `include "my_driver.svh"
  `include "my_monitor.svh"

```

```

`include "my_coverage.svh"
`include "my_agent.svh"
endpackage

```

## Agent

```

// my_agent.svh

class my_agent extends uvm_agent;
  typedef uvm_sequencer #(my_item) sequencer_t;
  typedef my_driver   driver_t;
  typedef my_monitor  monitor_t;
  typedef my_coverage coverage_t;
  my_config      cfg;
  sequencer_t    sqr;
  driver_t       drv;
  monitor_t      mon;
  coverage_t     cov;

  function void build();
    ...
    if (cfg.is_active) begin
      sqr=sequencer_t::type_id::create("sqr",this);
      drv=driver_t::type_id::create("drv",this);
    end
    mon=monitor_t::type_id::create("mon",this);
    if (cfg.has_coverage)
      cov = coverage_t::type_id::create("cov", this);
  endfunction

  function void connect();
    ...
    if (cfg.is_active) drv.seq_item_port.connect(sqr.seq_item_export);
    if (cfg.has_coverage) mon.o_my.connect(cov.analysis_export);
  endfunction

endclass

```

## Interface

```

// my_interface.svh

interface my_interface (clock, reset);
  input clock;
  input reset;

```

```
endinterface
```

## Data & Sequences

```
// my_data.svh

package my_data;
  class my_item extends uvm_sequence_item;
    ...
  endclass
endpackage

// my_sequence.svh

class my_sequence extends uvm_sequence #(my_item);
  //
endclass
```

## Driver

```
// my_driver.svh

class my_driver extends uvm_driver #(my_item);
  my_config cfg;
  task run();
    forever begin
      my_item it;
      seq_item_port.get_next_item(it);
      ... drive it
      seq_item_port.item_done();
    end
  endtask
endclass
```

## Monitor

```
// my_monitor.svh

class my_monitor extends uvm_monitor;
  my_config cfg;
  uvm_analysis_port #(my_item) o_my;

  function void build();
    super.build();
    o_my = new("o_my",this);
  endfunction
```

```
task run();
  forever begin
    my_item it;
    ... monitor it
    o_my.write(it);
  end
endtask
endclass
```

## Coverage

```
// my_coverage.svh

class my_coverage extends uvm_subscriber #(my_item);
  my_config cfg;
  my_item tx;

  covergroup my_cov;
    option.per_instance = 1;
    // Transaction Fields:
  endgroup

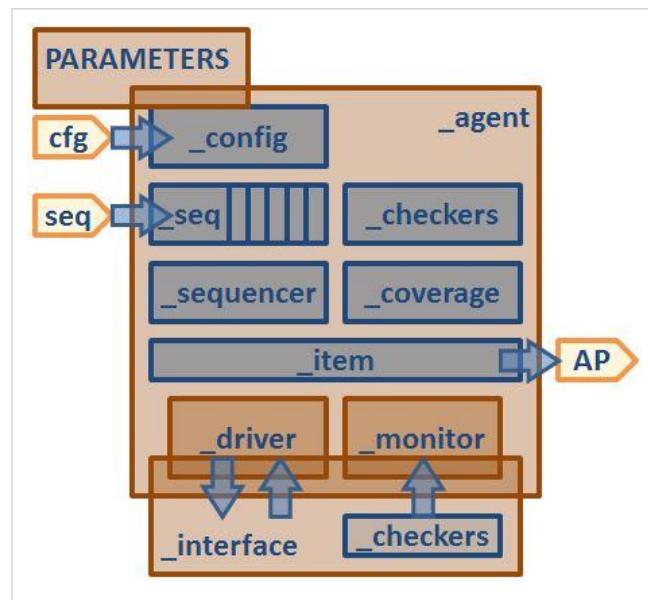
  function new(string name="my_coverage",uvm_component parent=null);
    super.new(name,parent);
    my_cov = new();
  endfunction

  function void write(my_item t);
    this.tx = t;
    my_cov.sample();
  endfunction

endclass
```

## Example parameterized Protocol UVC code

If the interface is parameterized, then the virtual interface and components which use it must also be parameterized (driver, monitor, agent, enclosing env). Parameterize by type to simplify this, as discussed in ParametersAndReuse. Sometimes, the coverage object may need to be parameterized too.



```

class my_agent #(type params=int) extends uvm_agent;
    typedef uvm_sequencer #(my_item) sequencer_t;
    typedef my_driver #(params) driver_t;
    typedef my_monitor #(params) monitor_t;
    typedef my_coverage #(params) coverage_t;
    my_config      cfg;
    sequencer_t    sqr;
    driver_t       drv;
    monitor_t      mon;
    coverage_t     cov;
    ...
endclass

interface my_interface #(type P=int) (clock, reset);
    ...
endinterface

class my_driver #(type params=int) extends uvm_driver #(my_item);
    ...
endclass

class my_monitor #(type params=int) extends uvm_monitor;
    ...
endclass

class my_coverage #(type params=int) extends uvm_subscriber #(my_item);
    ...

```

```
endclass
```

## Recommended directory/file structure

It is desirable that UVCs follow a familiar file manifest and layout. Not always possible depending on the internal implementation. For conventional SV implementations, the following file layout is suggested:

- A directory named after the component and its version number, e.g. hdmi1.4\_uvc-2.0.3/
  - README/files
- A directory named doc/
  - documentation
- A directory named uvc/ containing the SV uvc code
- The following named files (some optional) containing code as above:
  - hdmi1.4\_uvc\_pkg.sv
  - hdmi1.4\_interface.svh
  - hdmi1.4\_data.svh
  - hdmi1.4\_config.svh
  - hdmi1.4\_sequence.svh
  - hdmi1.4\_driver.svh
  - hdmi1.4\_monitor.svh
  - hdmi1.4\_agent.svh
  - hdmi1.4\_coverage.svh

## Background and Definitions

Component (electronics)

A basic element in a discrete form with terminals, intended to be connected together with others to form a system.

Component (software)

A manifestation in code of the above paradigm, emphasizing the separation of concerns and the encapsulation of a set of related functions and data into a modular, cohesive unit.

Verification Component

A packaged set of definitions and behavior covering a specific aspect of DUT functionality which can be deployed as a unit in a verification environment. Provides the means to verify the covered functionality in isolation, and can participate with other such components in a higher level verification environment. Examples of scope would be a communication or bus interface, a standard protocol, or a specific internal functional block.

**Also known as:** Verification Component (VC), Verification IP (VIP), Transactor, [Insert Marketing Term Here] Verification Component (xVC), Bus Functional Model (BFM)

**Provides:** functionality, interfaces, rapid bringup, complete verification solution, automatic stimulus generation, assertion checking, functional coverage analysis

**Attributes:** reusable, extensible, configurable, pre-verified, plug-and-play, independent, self-sufficient, substitutable, ease of use, embody a methodology, familiar

**Structure:** has means to communicate with other TB components, has means to communicate with DUT interfaces, participates in common elaborate/simulate phases, object extension of a base class

## Aspects of definition of a UVC

- from a verification/marketing point of view:  
how complete a solution it is and how it can interact with other components for reuse and further checking/scoreboarding
- at the file/directory and SV-package level:  
its packaging, compile/elaboration and integration flow/expectations, and familiarity for users out of the box
- from a SV/UVM coding point of view:  
its class hierarchy and interfaces, how it is intended to be configured and used, how it can be hooked up

## Attributes of a UVC

- relevant to native verification of the protocol or function in question
- familiar artifact for teams who know UVM to integrate, modify, replicate, reuse.
- conformant to the set of de facto standards of methodology around UVM
- consistent with the intent of the UVM methodology
- compatible with the UVM base class API and semantics

# Package/Organization

## Introduction

UVM organizes all of its base classes into a SystemVerilog Package. Code that is written extending from the UVM base classes should also be placed into packages in a specific hierarchy. The organization of this hierarchy makes it easier to compile smaller pieces of a testbench at once and in order and prevents circular references.

## Package Definitions

### Agent Packages

At the bottom of the package hierarchy agent packages provide the basic building blocks for a UVM testbench. An agent package contains a single import of the uvm\_pkg, possibly some typedefs and several `includes for files needed to make that agent work. These `include files include the definition of the sequence\_item type that the agent works with, the config object used to configure the agent, the components such as the driver and monitor needed to communicate with the bus and possibly some API Sequences. An example agent package definition follows:

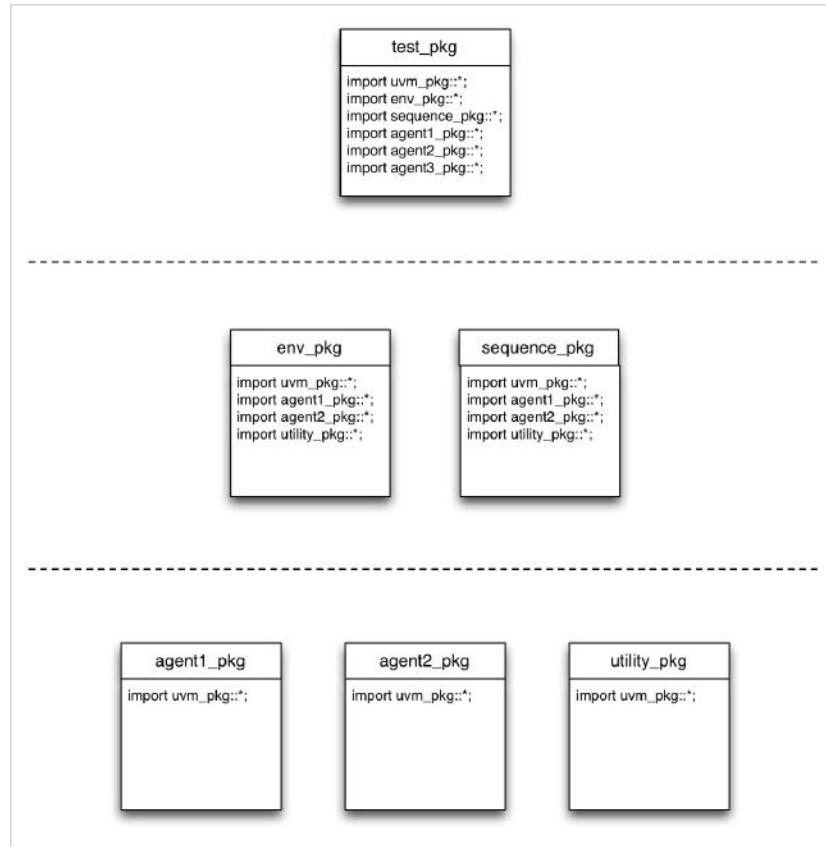
```
/*
 * This package contains all of the components, transactions
 * and sequences related to the mem_agent. Import this
 * package if you need to use the mem_agent anywhere.
 */

package mem_agent_pkg;

import uvm_pkg::*;

`include "uvm_macros.svh"

//Typedef used by the mem_agent
```



```
typedef enum bit[1:0] {READ, WRITE, PAUSE} instruction_type;

//Include the sequence_items (transactions)
`include "mem_item.svh"

//Include the agent config object
`include "mem_config.svh"

//Include the components
`include "mem_driver.svh"
`include "mem_monitor.svh"
//Create a typedef for the mem_sequencer since the
// uvm_sequencer class should not be extended
typedef uvm_sequencer #(mem_item) mem_sequencer;
`include "mem_trans_recorder.svh"
`include "mem_agent.svh"

//Include the API sequences
`include "mem_seq_base.svh"
`include "mem_read.svh"
`include "mem_write.svh"

endpackage : mem_agent_pkg
```

## Utility Packages

In addition to agent packages at the bottom of the hierarchy, there may also be utility packages. Utility packages contain definitions of useful classes, types, DPI function/task, etc. An example utility package that most people would be familiar with would be a register model. A Test param package would be another example of a utility package.

## Environment Package

The environment package will import the uvm\_pkg as well as any agent and any utility packages needed. The agent packages are imported because they contain the definitions of the agent classes and the configuration object classes which the environment will need to instantiate and work with. Utility packages are imported because they also contain class definitions, DPI function/task definitions, etc. that are needed. Several files are `included into the environment package. The environment class definition is `included along with analysis components such as scoreboards, test predictors/models and coverage classes. If there are a large number of analysis components they may be broken out into their own package which would then be imported into the environment package. An example environment package follows:

```
*****
/* This package contains the fixed part of the environment, */
/* the part that does not change for each testcase */
*****
```

```
package environment_pkg;
  import uvm_pkg::*;
  import mem_agent_pkg::*;
  import ahb_agent_pkg::*;

`include "uvm_macros.svh"

`include "scoreboard.svh"
`include "test_predictor.svh"
`include "coverage.svh"
`include "analysis_group.svh"
`include "environment.svh"

endpackage : environment_pkg
```

## Sequence Packages

Sequence packages also will need to import the uvm\_pkg and any agent packages needed to understand the sequence\_items they are going to create. If register sequences are being created using a UVM register model, then the register model package would also be imported. The files that would be `included into a sequence package would be any sequence base classes and the sequences themselves. The sequence package may and usually will be split up into multiple packages which will import each other. The divisions of when to create a new package follow the same divisions layed out in the Sequences/Hierarchy article. This means that usually the testbench will have multiple worker sequence packages which would all then be imported by a top level virtual sequence package. An example worker sequence package follows:

```
package sequence_pkg;
  import uvm_pkg::*;
  import bus_agent_pkg::*;
  import dut_registers_pkg::*;

`include "uvm_macros.svh"

  //Sequences
`include "no_reg_model_seq.svh"
`include "reg_model_seq.svh"
`include "reg_backdoor_seq.svh"
`include "mem_seq.svh"

endpackage : sequence_pkg
```

## Test Package

At the top of the package hierarchy is the test package. As the diagram shows, the test package will generally import all of the other packages. This is due to the test requiring understanding of everything that is going on in the testbench/environment. The test package will `include any base test classes and any tests derived from the base test classes. An example test package follows:

```
//-----
// test_pkg
//-----

package test_pkg;

`include "uvm_macros.svh"

import uvm_pkg::*;
import sequence_pkg::*;
import bus_agent_pkg::*;
import environment_pkg::*;
import dut_registers_pkg::*;

`include "test_base.svh"
`include "no_reg_model_test.svh"
`include "reg_model_test.svh"
`include "reg_backdoor_test.svh"
`include "mem_test.svh"

endpackage: test_pkg
```

## Package Compilation

With the organization layed out here, the testbench code does not contain any circular references. It also allows for staged, independent compiles of the different levels of packages. This means agent and utility packages can be compiled independently after the uvm\_pkg is compiled since they only import the uvm\_pkg. This allows for quicker compile checks when writing new code since only the package which contains changed files must be recompiled. Once all of the agent and utility packages are compiled, then the next level of packages the env\_pkg and the sequence\_pkg can be compiled. Finally the test\_pkg can be compiled, but only after we have successfully compiled everything it requires. Using this organization results in very structured makefiles and compilation scripts. An snippet from an example makefile follows.

```
DEPTH      = ..
DUT_HOME   = $(DEPTH)/dut
DUT_FILES  = $(DUT_HOME)/dut.sv
REG_MODEL_HOME = $(DEPTH)/register_files
REG_MODEL_FILES = $(REG_MODEL_HOME)/dut_registers_pkg.sv
BUS_AGT_HOME = $(DEPTH)/sv/bus_agent
BUS_AGT_FILES = $(BUS_AGT_HOME)/bus_agent_pkg.sv \
```

```
          $(BUS_agt_HOME)/bus_interface.sv
SEQS_HOME      = $(DEPTH)/sequences
SEQS_FILES     = $(SEQS_HOME)/sequence_pkg.sv
TB_HOME        = $(DEPTH)/tb
TB_FILES       = $(TB_HOME)/environment_pkg.sv
TESTS_HOME     = $(DEPTH)/tests
TESTS_FILES    = $(TESTS_HOME)/test_pkg.sv
TOP_MODULE     = $(TB_HOME)/testbench.sv

compile_reg_model_pkg:
  vlog -incr \
    +incdir+$ (REG_MODEL_HOME) \
    $ (REG_MODEL_FILES)

compile_bus_agent_pkg: compile_reg_model_pkg
  vlog -incr \
    +incdir+$ (BUS_agt_HOME) \
    $ (BUS_agt_FILES)

compile_sequences_pkg: compile_bus_agent_pkg
  vlog -incr \
    +incdir+$ (SEQS_HOME) \
    $ (SEQS_FILES)

compile_tb_files: compile_sequences_pkg
  vlog -incr \
    +incdir+$ (TB_HOME) \
    $ (TB_FILES)

compile_tests: compile_tb_files
  vlog -incr \
    +incdir+$ (TESTS_HOME) \
    $ (TESTS_FILES)

compile: compile_tests
  vlog -incr \
    $ (DUT_FILES) \
    $ (TOP_MODULE)
```

## Potential Pitfalls

Using this package organization technique and compilation strategy avoids numerous problems. This includes type mis-matches and circular references.

### "I'm sure it's the same type. I `included the class right here to be sure."

When a file is `included in SystemVerilog, it is as if the contents of the file were copied and pasted into the location of the `include. This is handy, but also dangerous especially when defining a class. In SystemVerilog, a fully qualified class type includes both the name class name and the scope the class is defined in. When a class is `included into a package, the fully qualified type name for the class is package\_name::class\_name. For example if we have an agent1\_seq\_item class `included into the agent1\_pkg from above, the fully qualified type name is agent1\_pkg::agent1\_seq\_item. This is important because if someone decided that they needed to understand what an agent1\_seq\_item class was and they just `included the agent1\_seq\_item.svh file into the location they needed the class definition, they have now created a new separate type which is not type compatible with the original type. This is why the class is `included exactly one time into a single package and the package is then imported where needed to preserve visibility. This problem is described in a blog post titled SystemVerilog Coding Guidelines: Package import versus `include<sup>[1]</sup> which provides additional information and examples.

### "Questa can't find the type of a class I'm using."

More than likely the package that the class was `included into wasn't imported into the package or module where the class type was trying to be used. Simply add an import statement to the top of the package or module to expose the class definition to the scope of the package or module.

### "run\_test() can't find my test. I know it's in my test\_pkg."

Even if the tests are `included properly into your test\_pkg, the test\_pkg will not be loaded into the simulation unless someone imports it. If the test\_pkg is not imported into the simulation, then tests will not be registered with the Factory. Since run\_test() uses the Factory to create the test it must know about the test that it is being asked to create. The solution to this is to import the test\_pkg into the top level testbench module.

## Directory Structure

Every package should be in its own directory. This also includes all the files that will be `included into the package with the exception of the "uvm\_macros.svh" file which will be `included with every package. After putting every package into its own directory, then other optional organization can take place. We suggest putting all agent package directories into higher level directory. All of the sequence package directories also generally will be grouped together into a higher level directory. Following is an example directory structure. The important thing is to be organized within your group/company, not necessarily to exactly follow what is shown.

```
▽ □ sv
  ▷ □ input_agent
  ▷ □ output_agent
  ▷ □ white_box_agent
  ▷ □ tb
  ▷ □ tests
```

# Appendix - Coding Guidelines

## SV/Guidelines

### Mentor Graphics SystemVerilog Guidelines

SystemVerilog Do's	
<ul style="list-style-type: none"> <li>• Use a consistent coding style - see guidelines</li> <li>• Use a descriptive typedef for variables</li> <li>• Use an end label for methods, classes and packages</li> <li>• Use `includes to compile classes into packages</li> <li>• Define classes within packages</li> <li>• Define one class per file</li> <li>• Only `include a file in one package</li> <li>• Import packages to reference their contents</li> <li>• Check that \$cast() calls complete successfully</li> <li>• Check that randomize() calls complete successfully</li> <li>• Use if rather than assert to check the status of method calls</li> <li>• Wrap covergroups in class objects</li> <li>• Only sample covergroups using the sample() method</li> <li>• Label covergroup coverpoints and crosses</li> </ul>	
SystemVerilog Don'ts	
<ul style="list-style-type: none"> <li>• Avoid `including the same class in multiple locations</li> <li>• Avoid placing code in \$unit</li> <li>• Avoid using associative arrays with a wildcard index</li> <li>• Avoid using #0 delays</li> <li>• Don't rely on static initialization order</li> </ul>	

The SystemVerilog coding guidelines and rules in this article are based on Mentor's experience and are designed to steer users away from coding practices that result in SystemVerilog that is either hard to understand or debug.

Please send any suggestions, corrections or additions to ?subject=SV/Guidelines vmdoc@mentor.com <sup>[1]</sup>

### General Coding Style

Although bad coding style does not stop your code from working, it does make it harder for others to understand and makes it more difficult to maintain. Take pride in writing well-ordered and uniformly formatted code.

#### 1.1 Guideline: Indent your code with spaces

Use a consistent number of spaces to indent your code every time you start a new nested block, 2 or 3 spaces is recommended. Do not use tabs since the tab settings vary in different editors and viewers and your formatting may not look as you intended. Many text editors have an indenting mode that automatically replaces tabs with a defined number of spaces.

#### 1.2 Guideline: Only one statement per line

Only have one declaration or statement per line. This makes the code clearer and easier to understand and debug.

Recommended	Not Recommended
<pre>// Variable definition: logic enable; logic completed; logic in_progress;  // Statements: // (See next Guideline for the use of begin-end pairs // with conditional statements) // if(enable == 0)     in_progress = 1; else     in_progress = 0;</pre>	<pre>// Variable definition: logic enable, completed, in_progress;  // Statements: if(enable == 0) in_progress = 1; else in_progress = 0;</pre>

### 1.3 Guideline: Use a begin-end pair to bracket conditional statements

This helps make it clear where the conditional code begins and where it ends. Without a begin-end pair, only the first line after the conditional statement is executed conditionally and this is a common source of errors.

Recommended	Not Recommended
<pre>// Both statements executed conditionally: if(i &gt; 0)     begin         count = current_count;         target = current_target;     end</pre>	<pre>if(i &gt; 0)     count = current_count;     target = current_target; // This statement is executed unconditionally</pre>

### 1.4 Guideline: Use parenthesis in Boolean conditions

This makes the code easier to read and avoids mistakes due to operator precedence issues.

Recommended	Not Recommended
<pre>// Boolean or conditional expression if((A==B) &amp;&amp; (B &gt; (C*2))    (B &gt; ((D**2)+1))) begin     ... end</pre>	<pre>// Boolean or conditional expression if(A==B &amp;&amp; B &gt; C*2    B &gt; D**2+1) begin     ... end</pre>

### 1.5 Guideline: Keep your code simple

Avoid writing tricky and hard to understand code, keep it simple so that it is clear what it does and how so that others can quickly understand it in case a modification is required.

### 1.6 Guideline: Keep your lines to a reasonable length

Long lines are difficult to read and understand, especially if you need to scroll the editor to the right to read the end of the line. As a guideline, keep your line length to around 80 characters, break the line and indent at logical places.

<b>Not Recommended</b>	<pre> function bit do_compare(uvm_object rhs, uvm_comparer comparer);     mbus_seq_item rhs_;      if(!\$cast(rhs_, rhs)) begin         uvm_report_error("do_compare", "cast failed, check type compatibility");         return 0;     end     do_compare = super.do_compare(rhs, comparer) &amp;&amp; (MADDR == rhs_.MADDR) &amp;&amp; (MWDATA == rhs_.MWDATA) &amp;&amp; (MREAD == rhs_.MREAD) &amp;&amp; (MOPCODE == rhs_.MOPCODE) &amp;&amp; (MPHASE == rhs_.MPHASE) &amp;&amp; (MRESP == rhs_.MRESP) &amp;&amp; (MRDATA == rhs_.MRDATA); endfunction: do_compare </pre>
<b>Recommended</b>	<pre> function bit do_compare(uvm_object rhs, uvm_comparer comparer);     mbus_seq_item rhs_;      if(!\$cast(rhs_, rhs)) begin         uvm_report_error("do_compare", "cast failed, check type compatibility");         return 0;     end     do_compare = super.do_compare(rhs, comparer) &amp;&amp;         (MADDR == rhs_.MADDR) &amp;&amp;         (MWDATA == rhs_.MWDATA) &amp;&amp;         (MREAD == rhs_.MREAD) &amp;&amp;         (MOPCODE == rhs_.MOPCODE) &amp;&amp;         (MPHASE == rhs_.MPHASE) &amp;&amp;         (MRESP == rhs_.MRESP) &amp;&amp;         (MRDATA == rhs_.MRDATA); endfunction: do_compare </pre>

### 1.7 Guideline: Use lowercase for names, using underscores to separate fields

This makes it clearer what the name is, as opposed to other naming styles such as CamelCase which are harder to read.

Recommended	Not Recommended
axi_fabric_scoreboard_error	AxiFabricScoreboardError

### 1.8 Guideline: Use prefix\_ and \_postfix to delineate name types

Use prefixes and postfixes for name types to help differentiate between variables. Pre and post fixes for some common variable types are summarised in the following table:

prefix/postfix	Purpose
_t	Used for a type created via a typedef
_e	Used to indicate an enumerated type
_h	Used for a class handle
_m	Used for a protected class member (See guideline 2.2)
_cfg	Used for a configuration object handle
_ap	Used for an analysis port handle
_group	Used for a covergroup handle

### 1.9 Guideline: Use a descriptive typedef when declaring a variable instead of a built-in type

This makes the code clearer and easier to understand as well as easier to maintain. An exception is when the built-in type keyword best describes the purpose of the variable's type.

```
// Descriptive typedef for a 24 bit audio sample:  
typedef bit[23:0] audio_sample_t;
```

### 1.10 Guideline: Use the end label for classes, functions, tasks, and packages

This forces the compiler to check that the name of the item matches the end label which can trap cut and paste errors. It is also useful to a person reading the code.

```
// Using end labels  
package my_pkg;  
  
//...  
class my_class;  
  
// ...  
function void my_function();  
//...  
endfunction: my_function  
  
task my_task;  
// ...  
endtask: my_task  
  
endclass: my_class  
  
endpackage: my_pkg
```

### 1.11 Guideline: Comment the intent of your code

Add comments to define the intent of your code, don't rely on the users interpretation. For instance, each method in a class should have a comment block that specifies its input arguments, its function and its return arguments.

This principle can be extended to automatically generate html documentation for your code using documentation tools such as **NaturalDocs**.

## Class Names and Members

### 2.1 Guideline: Name classes after the functionality they encapsulate

Use classes to encapsulate related functionality. Name the class after the functionality, for instance a scoreboard for an Ethernet router would be named "router\_scoreboard".

### 2.2 Guideline: Private class members should have a m\_ prefix

Any member that is meant to be private should be named with a 'm\_' prefix, and should be made local or protected. Any member that will be randomized should not be local or protected.

### 2.3 Guideline: Declare class methods using extern

This means that the class body contains the method prototypes and so users only have to look at this section of the class definition to understand its functionality.

```
// Descriptive typedefs:  
typedef logic [31:0] raw_sample_t;  
typedef logic [15:0] processed_sample_t  
  
// Class definition illustrating the use of externally defined methods:  
class audio_compress;  
  
rand int iteration_limit;  
rand bit valid;  
rand raw_sample_t raw_audio_sample;  
rand processed_sample_t processed_sample;  
  
// function: new  
// Constructor - initializes valid  
extern function new();  
  
// function: compress_sample  
// Applies compression algorithm to raw sample  
// inputs: none  
// returns: void  
extern function void compress_sample();  
  
// function: set_new_sample  
// Set a new raw sample value
```

```

// inputs:
//   raw_sample_t new_sample
// returns: void
extern function void set_new_sample(raw_sample_t new_sample);

endclass: audio_compress

function audio_compress::new();
  valid = 0;
  iteration_limit = $bits(processed_sample_t);
endfunction

function void audio_compress::compress_sample();
  for(int i = 0; i < iteration_limit; i++) begin
    processed_sample[i] = raw_audio_sample[((i*2)-1):(i*2)];
  end
  valid = 1;
endfunction: compress_sample

function void audio_compress::set_new_sample(raw_sample_t new_sample);
  raw_audio_sample = new_sample;
  valid = 0;
endfunction: set_new_sample

```

## Files and Directories

The following guidelines concern best practices for SystemVerilog files and directories.

### File Naming

**3.1 Guideline: Use lower case for file and directory names**

Lower case names are easier to type.

**3.2 Guideline: Use .sv extensions for compile files, .svh for `include files**

The convention of using the .sv extension for files that are compiled and .svh for files that get included makes it easier to sort through files in a directory and also to write compilation scripts.

For instance, a package definition would have a .sv extension, but would reference `included .svh files:

**3.3 Guideline: `include .svh class files should only contain one class and be named after that class**

This makes it easier to maintain the code, since it is obvious where the code is for each class.

**3.4 Guideline: Use descriptive names that reflect functionality**

File names should match their content. The names should be descriptive and use postfixes to help describe the intent - e.g. \_pkg, \_env, \_agent etc.

## `include versus import

### 3.5 Rule: Only use `include to include a file in one place

The `include construct should only be used to include a file in just one place. `include is typically used to include .svh files when creating a package file.

If you need to reference a type or other definition, then use 'import' to bring the definition into scope. Do not use `include. The reason for this is that type definitions are scope specific. A type defined in two scopes using the same `include file are not recognised as being the same. If the type is defined in one place, inside a package, then it can be properly referenced by importing that package.

An exception to this would be a macro definition file such as the 'uvm\_macros.svh' file.

## Directory Names

Testbenches are constructed of SystemVerilog UVM code organized as **packages**, collections of verification IP organized as packages and a description of the hardware to be tested. Other files such as C models and documentation may also be required. Packages should be organized in a hierarchy.

### 3.6 Guideline: Each package should have its own directory

Each package should exist in its own directory. Each of these package directories should have one file that gets compiled - a file with the extension .sv

Each package should have at most one file that may be included in other code. This file may define macros.

```
abc_pkg.sv
abc_macros.svh
```

For a complex package (such as a UVC) that may contain tests, examples and documentation, create subdirectories:

```
abc_pkg/examples
abc_pkg/docs
abc_pkg/tests
abc_pkg/src/abc_pkg.sv
```

For a simple package the subdirectories may be omitted

```
abc_pkg/abc_pkg.sv
```

## Sample File Listing

```
./abc_pkg/src
./abc_pkg/src/abc_pkg.sv

./abc_pkg/src/abc_macros.svh

./abc_pkg/src/abc_env.svh
```

```

./abc_pkg/src/abc_interface.sv

./abc_pkg/src/abc_driver.svh
./abc_pkg/src/abc_monitor.svh
./abc_pkg/src/abc_scoreboard.svh

./abc_pkg/src/abc_sequence_item.svh
./abc_pkg/src/abc_sequencer.svh
./abc_pkg/src/abc_sequences.svh

./abc_pkg/docs/
./abc_pkg/docs/abc_user_guide.docx

./abc_pkg/tests/
./abc_pkg/tests/.......

./abc_pkg/examples/
./abc_pkg/examples/a/....
./abc_pkg/examples/b/....
./abc_pkg/examples/c/.....

./testbench1/makefile
./testbench1/tb_env.sv
./testbench1/tb_top.sv
./testbench1/test.sv

```

## Using Packages

### 3.7 Rule: Import packages to reference their contents

When you use a function or a class from a package, you import it, and `include any macro definitions.

If you `include the package source, then you will be creating a new namespace for that package in every file that you `include it into, this will result in type matching issues.

```

import abc_pkg::*;
`include "abc_macros.svh"

```

### 3.8 Rule: When compiling a package, use +includir+ to reference its source directory

To compile the package itself, you use a +includir+ to reference the source directory. Make sure that there are no hardcoded paths in the path string for the `included file.

```
vlog +includir+$ABC_PKG/src abc_pkg.sv
```

To compile code that uses the package, you also use a +includir+ to reference the source directory if a macro file needs to be `included.

```
vlog +incdir+$ABC_PKG/src tb_top.sv
```

To compile the packages, and the testbench for the example:

```
vlib work

# Compile the Questa UVM Package (for UVM Debug integration)
vlog +incdir+$QUESTA_UVM_HOME/src \
$QUESTA_UVM_HOME/src/questa_uvm_pkg.sv

# Compile the VIP (abc and xyz)
vlog +incdir+../abc_pkg/src \
..../abc_pkg/src/abc_pkg.sv
vlog +incdir+../xyz_pkg/src \
..../xyz_pkg/src/xyz_pkg.sv

# Compile the DUT (RTL)
vlog ..../dut/dut.sv

# Compile the test
vlog +incdir+../test_pkg/src \
..../test_pkg/src/test_pkg.sv

# Compile the top
vlog tb_top.sv

# Simulate
vsim -uvm=debug -coverage +UVM_TESTNAME=test \
-c tb_top -do "run -all; quit -f"
```

## SystemVerilog Language Guidelines

### 4.1 Rule: Check that \$cast() has succeeded

If you are going to use the result of the cast operation, then you should check the status returned by the \$cast call and deal with it gracefully, otherwise the simulation may crash with a null pointer.

Note that it is not enough to check the result of the cast method, you should also check that the handle to which the cast is made is not null. A cast operation will succeed if the handle from which the cast is being done is null.

```
// How to check that a $cast has worked correctly
function my_object get_a_clone(uvm_object to_be_cloned);
    my_object t;

    if(!$cast(t, to_be_cloned.clone())) begin
        `uvm_error("get_a_clone", "$cast failed for to_be_cloned")
```

```

    end

    if(t == null) begin
        `uvm_fatal("get_a_clone", "$cast operation resulted in a null handle, check to_be_cloned handle")
    end

    return t;
endfunction: get_a_clone

```

#### 4.2 Rule: Check that randomize() has succeeded

If no check is made the randomization may be failing, meaning that the stimulus generation is not working correctly.

```

// Using if() to check randomization result
if(!seq_item.randomize() with {address inside {[0:32'hF000_FC00]};}) begin
    `uvm_error("seq_name", "randomization failure, please check constraints")
end

```

#### 4.3 Rule: Use if rather than assert to check the status of method calls

Assert results in the code check appearing in the coverage database, which is undesired. Incorrectly turning off the action blocks of assertions may also produce undesired results.

### Constructs to be Avoided

The SystemVerilog language has been a collaborative effort with a long history of constructs *borrowed* from other languages. Some constructs have been improved upon with newer constructs, but the old constructs remain for backward compatibility and should be avoided. Other constructs were added before being proven out and in practice cause more problems than they solve.

#### 4.4 Rule: Do not place any code in \$unit, place it in a package

The compilation unit, \$unit, is the scope outside of a design element (package, module, interface, program). There are a number of problems with timescales, visibility, and re-usability when you place code in \$unit. Always place this code in a package.

#### 4.5 Guideline: Do not use associative arrays with a wildcard index[\*]

A wildcard index on an associative array is an un-sized integral index. SystemVerilog places severe restrictions on other constructs that cannot be used with associative arrays having a wildcard index. In most cases, an index type of [int] is sufficient. For example, a foreach loop requires a fixed type to declare its iterator variable.

```

string names[*]; // cannot be used with foreach, find_index, ...
string names[int];
...
foreach (names[i])
    $display("element %0d: %s", i, names[i]);

```

#### 4.6 Guideline: Do not use #0 procedural delays

Using a #0 procedural delay, sometimes called a delta delay, is a sure sign that you have coded incorrectly. Adding a #0 just to get your code working usually avoids one race condition and creates another one later. Often, using a non-blocking assignment (`<=`) solves this class of problem.

#### 4.7 Guideline: Avoid the use of the following language constructs

A number of SystemVerilog language constructs should be avoided altogether:

Construct	Reason to avoid
checker	Ill defined, not supported by Questa
final	Only gets called when a simulation completes
program	Legacy from Vera, alters timing of sampling, not necessary and potentially confusing

## Coding Patterns

Some pieces of code fall into well recognized patterns that are known to cause problems

#### 4.8 Rule: Do not rely on static variable initialization order, initialize on first instance.

The ordering of static variable initialization is undefined. If one static variable initialization requires the non-default initialized value of another static variable, this is a race condition. This can be avoided by creating a static function that initializes the variable on the first reference, then returns the value of the static variable, instead of directly referencing the variable.

```

typedef class A;
typedef class B;
A a_top=A::get_a();
B b_top=B::get_b();
class A;
    static function A get_a();
        if (a_top == null) a_top =new();
        return a_top;
    endfunction
endclass : A
class B;
    A a_top;
    protected function new();
        a_top = get_a();
    endfunction
    static function B get_b();
        if (b_top == null) b_top =new();
        return b_top;
    endfunction
endclass : B

```

## Covergroups

### 4.9 Guideline: Create covergroups within wrapper classes

Covergroups have to be constructed within the constructor of a class. In order to make the inclusion of a covergroup within a testbench conditional, it should be wrapped within a wrapper class.

### 4.10 Guideline: Covergroup sampling should be conditional

Build your covergroups so that their sample can be turned on or off. For example use the 'iff' clause of covergroups.

```
// Wrapped covergroup with sample control:  
class cg_wrapper extends uvm_component;  
  
logic[31:0] address;  
bit coverage_enabled  
  
covergroup detail_group;  
  ADDRESS: coverpoint addr iff(coverage_enabled) {  
    bins low_range = {[0:32'h0000_ffff]};  
    bins med_range = {[32'h0001_0000:32'h0200_ffff]};  
    bins high_range = {[32'h0201_0000:32'h0220_ffff]};  
  }  
// ....  
endgroup: detail_group  
  
function new(string name = "cg_wrapper", uvm_component parent = null);  
  super.new(name, parent);  
  // Construct covergroup and enable sampling  
  detail_group = new();  
  coverage_enabled = 1;  
endfunction  
  
// Set coverage enable bit - allowing coverage to be enabled/disabled  
function void set_coverage_enabled(bit enable);  
  coverage_enabled = enable;  
endfunction: set_coverage_enabled  
  
// Get current state of coverage enabled bit  
function bit get_coverage_enabled();  
  return coverage_enabled;  
endfunction: get_coverage_enabled  
  
// Sample the coverage group:  
function void sample(logic[31:0] new_address);  
  address = new_address;  
endfunction
```

```
    detail_group.sample();
endfunction: sample
```

Coverpoint sampling may not be valid in certain situations, for instance during reset.

```
// Using iff to turn off unnecessary sampling:

// Only sample if reset is not active
coverpoint data iff(reset_n != 0) {
    // Only interested in high_end values if high_pass is enabled:
    bins high_end = {[10000:20000]} iff(high_pass);
    bins low_end = {[1:300]};
}
```

## Collecting Coverage

### 4.11 Guideline: Use the covergroup sample() method to collect coverage

Sample a covergroup by calling the sample routine, this allows precise control on when the sampling takes place.

### 4.12 Rule: Label coverpoints and crosses

Labelling coverpoints allows them to be referenced in crosses and easily identified in reports and viewers.

```
payload_size_cvpt: coverpoint ...
```

Labelling crosses allows them to be easily identified

```
payload_size_X_parity: cross payload_size_cvpt, parity;
```

### 4.13 Guideline: Name your bins

Name your bins, do not rely on auto-naming.

```
bin minimum_val = {min};
```

### 4.14 Guideline: Minimize the size of the sample

It is very easy to specify large numbers of bins in covergroups through autogeneration without realising it. You can minimise the impact of a covergroup on simulation performance by thinking carefully about the number and size of the bins required, and by reducing the cross bins to only those required.

## Other SystemVerilog Guidelines Documents

- Stu Sutherlands' SystemVerilog for Design <sup>[2]</sup>
- Chris Spear's SystemVerilog for Verification <sup>[3]</sup>
- Doulos' SystemVerilog Golden Reference Guide <sup>[4]</sup>
- Adam Erickson's Are Macros Evil? DVCon 2011 Best Paper <sup>[1]</sup>

# UVM/Guidelines

---

## Mentor Graphics UVM Guidelines

UVM Do's
<ul style="list-style-type: none"><li>• Define classes within packages</li><li>• Define one class per file</li><li>• Use factory registration macros</li><li>• Use message macros</li><li>• Manually implement do_copy(), do_compare(), etc.</li><li>• Use sequence.start(sequencer)</li><li>• Use start_item() and finish_item() for sequence items</li><li>• Use the uvm_config_db API</li><li>• Use a configuration class for each agent</li><li>• Use phase objection mechanism</li><li>• Use the phase_ready_to_end() func</li><li>• Use the run_phase() in transactors</li><li>• Use the reset/configure/main/shutdown phases in tests</li></ul>
UVM Don'ts
<ul style="list-style-type: none"><li>• Avoid `including a class in multiple locations</li><li>• Avoid constructor arguments other than name and parent</li><li>• Avoid field automation macros</li><li>• Avoid uvm_comparer policy class</li><li>• Avoid the sequence list and default sequence</li><li>• Avoid the sequence macros (`uvm_do)</li><li>• Avoid pre_body() and post_body() in a sequence</li><li>• Avoid explicitly consuming time in sequences</li><li>• Avoid set/get_config_string/_int/_object()</li><li>• Avoid the uvm_resource_db API</li><li>• Avoid callbacks</li><li>• Avoid user defined phases</li><li>• Avoid phase jumping and phase domains (for now)</li><li>• Avoid raising and lowering objections for every transaction</li></ul>

The UVM library is both a collection of classes and a methodology for how to use those base classes. UVM brings clarity to the SystemVerilog language by providing a structure for how to use the features in SystemVerilog. However, in many cases UVM provides multiple mechanisms to accomplish the same work. This guideline document is here to provide some structure to UVM in the same way that UVM provides structure to the SystemVerilog language.

Mentor Graphics has also documented pure SystemVerilog Guidelines as well. Please visit the SV/Guidelines article for more information.

## Class Definitions

### 1.1 Rule: Define all classes within a package with the one exception of Abstract/Concrete classes.

Define all classes within a package. Don't `include class definitions haphazardly through out the testbench. The one exception to defining all classes within a package is Abstract/Concrete classes. Having all classes defined in a package makes it easy to share class definitions when required. The other way to bring in class definitions into an UVM testbench is to try to import the class wherever it is needed. This has potential to define your class multiple times if the class is imported into two different packages, modules, etc. If a class is `included into two different scopes, then SystemVerilog states that these two classes are different types.

Abstract/Concrete classes are the exception because they must be defined within a module to allow them to have access to the scope of the module. The Abstract/Concrete class is primarily used when integrating verification IP written in Verilog or VHDL.

### 1.2 Rule: Define one class per file and name the file <CLASSNAME>.svh.

Every class should be defined in its own file. The file should be named <CLASSNAME>.svh. The file should then be included in another file which defines a package. All files included into a single package should be in the same directory. The package name should end in \_pkg to make it clear that the design object is a package. The file that contains the class definition should not contain any import or include statements. This results in a file structure that looks like this:

example\_agent/ <-- Directory containing Agent code

example\_agent\_pkg.sv

example\_item.svh  
example\_config.svh  
example\_driver.svh  
example\_monitor.svh  
example\_agent.svh  
example\_api\_seq1.svh  
reg2example\_adapter.svh

With that list of files, the example\_pkg.sv file would look like this:

```
// Begin example_pkg.sv file
`include "uvm_macros.svh"

package example_pkg;
    import uvm_pkg::*;
    import another_pkg::*;

    //Include any transactions/sequence_items
    `include "example_item.svh"

    //Include any configuration classes
    `include "example_config.svh"

    //Include any components
    `include "example_driver.svh"
```

```

`include "example_monitor.svh"
`include "example_agent.svh"

//Include any API sequences
`include "example_api_seql.svh"

//Include the UVM Register Layer Adapter
`include "reg2example_adapter.svh"

endpackage : example_pkg
// End example_pkg.sv file

```

If one of the files that defines a class (example\_driver.svh) contains a package import statement in it, it would be just like the import statement was part of the example package. This could result in trying to import a package multiple times which is inefficient and wastes simulation time.

### 1.3 Rule: Call super.new(name [,parent]) as the first line in a constructor.

Explicitly pass the required arguments to the super constructor. That way the intent is clear. Additionally, the constructor should only contain the call to super.new() and optionally any calls to new covergroups that are part of the class. The calls to new the covergroups must be done in the constructor according to the SystemVerilog LRM. All other objects should be built in the build\_phase() function for components or the beginning of the body() task for sequences.

### 1.4 Rule: Don't add extra constructor arguments other than name and parent.

Extra arguments to the constructor will at worst case result in the UVM Factory being unable to create the object that is requested. Even if the extra arguments have default values which will allow the factory to function, the extra arguments will always be the default values as the factory only passes along the name and parent arguments.

## Factory

The UVM Factory provides an easy, effective way to customize an environment without having to extend or modify the environment directly. To make effective use of the UVM Factory and to promote as much flexibility for reuse of code as possible, Mentor Graphics recommends following guidelines. For more information, refer to the Factory article.

### 2.1 Rule: Register and create all classes with the UVM Factory.

Registering all classes and creating all objects with the UVM Factory maximizes flexibility in UVM testbenches. Registering a class with the UVM Factory carries no run-time penalty and only slight overhead for creating an object via the UVM Factory. Classes defined by UVM are registered with the factory and objects created from those classes should be created using the UVM Factory. This includes classes such as the uvm\_sequencer.

### 2.2 Rule: Import packages that define classes registered with the UVM Factory.

This guideline may seem obvious, but be sure to import all packages that have classes defined in them. If the package is not imported, then the class will not be registered with the UVM Factory. The most common place that this mistake is made is not importing the package that contains all the tests into the top level testbench module. If that test package is

not imported, then UVM will not understand the definition of the test the call to run\_test() attempts to create the test object.

**2.3 Guideline: When creating an object with the UVM Factory, match the object's handle name with the string name passed into the create() call.**

UVM builds an object hierarchy which is used for many different functions including UVM Factory overrides and configuration. This hierarchy is based on the string name that is passed into the first argument of every constructor. Keeping the handle name and this string hierarchy name the same will greatly aid in debug when the time comes. For more info, visit the Testbench/Build article.

## Macros

The UVM library contains many different types of macros. Some of them do a very small, well defined job and are very useful. However, there are other macros which may save a small bit of time initially, but will ultimately cost more time down the road. This extra time is consumed in both debug and run time. For more information on this topic, please see the MacroCostBenefit article.

### Factory Registration Macros

**3.1 Rule: Use the `uvm\_object\_utils(), `uvm\_object\_param\_utils(), `uvm\_component\_utils() and `uvm\_component\_param\_utils() factory registration macros.**

The factory registration macros provide useful, well defined functionality. As the name implies, these macros register the UVM object or component with the UVM factory which is both necessary and critical. These macros are `uvm\_object\_utils(), `uvm\_object\_param\_utils(), `uvm\_component\_utils() and `uvm\_component\_param\_utils(). When these macros are expanded, they provide the type\_id typedef (which is the factory registration), the static get\_type() function (which returns the object type), the get\_object\_type() function (which is not static and also returns the object type) and the create() function. Notice that the `uvm\_sequence\_utils() and `uvm\_sequencer\_utils() macros which also register with the factory are not recommended. Please see the starting sequences section for more information.

### Message Macros

**3.2 Guideline: Use the UVM message macros which are `uvm\_info(), `uvm\_warning(), `uvm\_error() and `uvm\_fatal().**

The UVM message macros provide a performance savings when used. The message macros are `uvm\_info(), `uvm\_warning(), `uvm\_error() and `uvm\_fatal(). These macros ultimately call uvm\_report\_info, uvm\_report\_warning, etc. What these macros bring to the table are a check to see if a message would be filtered before expensive string processing is performed. They also add the file and line number to the message when it is output.

## Field Automation Macros

**3.3 Guideline: Do write your own do\_copy(), do\_compare(), do\_print(), do\_pack() and do\_unpack() functions. Do not use the field automation macros.**

The field automation Macros on the surface look like a very quick and easy way to deal with data members in a class. However, the field automation macros have a very large hidden cost. As a result, Mentor Graphics does not use or recommend using these macros. These macros include `uvm\_field\_int(), `uvm\_field\_object(), `uvm\_field\_array\_int(), `uvm\_field\_queue\_string(), etc. When these macros are expanded, they result in hundreds of lines of code. The code that is produced is not code that a human would write and consequently very difficult to debug even when expanded. Additionally, these macros try to automatically get information from the UVM resource database. This can lead to unexpected behavior in a testbench when someone sets configuration information with the same name as the field being registered in the macro.

Mentor Graphics does recommend writing your own do\_copy(), do\_compare(), do\_print(), do\_pack() and do\_unpack() methods. Writing these methods out one time may take a little longer, but that time will be made up when running your simulations. For example, when writing your own do\_compare() function, two function calls will be executed to compare (compare() calls do\_compare() ) the data members in your class. When using the macros, 45 function calls are executed to do the same comparison. Additionally, when writing do\_compare(), do not make use of the uvm\_comparer policy class that is passed in as an argument. Just write your comparison to return a "0" for a mis-compare and a "1" for a successful comparison. If additional information is needed, it can be displayed in the do\_compare() function using the `uvm\_info(), etc. macros. The uvm\_comparer policy class adds a significant amount of overhead and doesn't support all types.

## Sequences

UVM contains a very powerful mechanism for creating stimulus and controlling what will happen in a testbench. This mechanism, called sequences, has two major use models. Mentor Graphics recommends starting sequences in your test and using the given simple API for creating sequence items. Mentor Graphics does not recommend using the sequence list or a default sequence and does not recommend using the sequence macros. For more information, please refer to the Sequences article.

### Starting Sequences

**4.1 Rule: Do start your sequences using sequence.start(sequencer).**

To start a sequence running, Mentor Graphics recommends creating the sequence in a test and then calling sequence.start(sequencer) in one of the run phase tasks of the test. Since start() is a task that will block until the sequence finishes execution, you can control the order of what will happen in your testbench by stringing together sequence.start(sequencer) commands. If two or more sequences need to run in parallel, then the standard SystemVerilog fork/join(\_any, \_none) pair can be used. Using sequence.start(sequencer) also applies when starting a child sequence in a parent sequence. You can also start sequences in the reset\_phase(), configure\_phase(), main\_phase() and/or the shutdown\_phase(). See Phasing below

## Using Sequence Items

### 4.2 Rule: Do create sequence items with the factory. Do use start\_item() and finish\_item(). Do not use the UVM sequence macros (`uvm\_do, etc.).

To use a sequence item in a sequence Mentor Graphics recommends using the factory to create the sequence item, the start\_item() task to arbitrate with the sequencer and the finish\_item() task to send the randomized/prepared sequence item to the driver connected to the sequencer.

Mentor Graphics does not recommend using the UVM sequence macros. These macros include the 18 macros which all begin with `uvm\_do, `uvm\_send, `uvm\_create, `uvm\_rand\_send. These macros hide the very simple API of using sequence.start() or the combination of start\_item() and finish\_item(). Instead of remembering three tasks, 18 macros have to be known if they are used and instances still exist where the macros don't provide the functionality that is needed. See the MacroCostBenefit article for more information.

## pre\_body() and post\_body()

### 4.3 Guideline: Do not use the pre\_body() and post\_body() tasks that are defined in a sequence.

Do not use the pre\_body() and post\_body() tasks that are defined in a sequence. Depending on how the sequence is called, these tasks may or may not be called. Instead put the functionality that would have gone into the pre\_body() task into the beginning of the body() task. Similarly, put the functionality that would have gone into the post\_body() task into the end of the body() task.

## Sequence Data Members

### 4.4 Guideline: Make sequence input variables rand, output variables non-rand.

Making input variables rand allows for either direct manipulation of sequence control variables or for easy randomization by calling sequence.randomize(). Data members of the sequence that are intended to be accessed after the sequence has run (outputs) should not be rand as this just would waste processor resources when randomizing. See the Sequences/API article for more info.

## Time Consumption

### 4.5 Rule: Sequences should not explicitly consume time.

Sequences should not have explicit delay statements (#10ns) in them. Having explicit delays reduces reuse and is illegal for doing testbench acceleration in an emulator. For more information on considerations needed when creating an emulation friendly UVM testbench, please visit the Emulation article.

## Virtual Sequences

**4.6 Guideline: When a virtual sequencer is used, virtual sequences should check for null sequencer handles before executing.**

When a virtual sequencer is used, a simple check to ensure a sequencer handle is not null can save a lot of debug time later.

## Phasing

UVM introduces a graph based phasing mechanism to control the flow in a testbench. There are several "build phases" where the testbench is configured and constructed. These are followed by "run-time phases" which consume time running sequences to cause the testbench to produce stimulus. "Clean-up phases" provide a place to collect and report on the results of running the test. See Phasing for more information.

**5.1 Guideline: Transactors should only implement the run\_phase(). They should not implement any of the other time consuming phases.**

Transactors (drivers and monitors) are testbench executors. A driver should process whatever transactions are sent to it from the sequencer and a monitor should capture the transactions it observes on the bus regardless of the when the transactions occur. See Phasing/Transactors for more information.

**5.2 Guideline: Avoid the usage of reset\_phase(), configure\_phase(), main\_phase(), shutdown\_phase() and the pre\_/post\_ versions of those phases.**

The reset\_phase(), configure\_phase(), main\_phase() or shutdown\_phase() phases will be removed in a future version of UVM. Instead to provide sync points (which is what the new phases essentially add), use normal fork/join statements to run sequences in parallel or just start sequences one after another to have sequences run in series. This works because sequences are blocking. For more advanced synchronization needs, uvm\_barriers, uvm\_events, etc. can be used.

**5.3 Guideline: Avoid phase jumping and phase domains (for now).**

These two features of UVM phasing are still not well understood. Avoid them for now unless you are a very advanced user.

**5.4 Guideline: Do not use user defined phases.**

Do not use user defined phases. While UVM provides a mechanism for adding a phase in addition to the defaults (build\_phase, connect\_phase, run\_phase, etc.), the current mechanism is very difficult to debug. Consider integrating and reusing components with multiple user defined phases and trying to debug problems in this scenario.

## Configuration

### **6.1 Rule: Use the uvm\_config\_db API to pass configuration information. Do not use set/get\_config\_object(), set/get\_config\_string() or set/get\_config\_int() as these are deprecated. Also do not use the uvm\_resource\_db API.**

UVM provides a mechanism for higher level components to configure lower level components. This allows a test to configure what a testbench will look like and how it will operate for a specific test. This mechanism is very powerful, but also can be very inefficient if used in an incorrect way. Mentor Graphics recommends using only the uvm\_config\_db API as it allows for any type and uses the component hierarchy to ensure correct scoping. The uvm\_config\_db API should be used to pass configuration objects to locations where they are needed. It should not be used to pass integers, strings or other basic types as it is much easier for a name space collision to happen when using low level types.

The set/get\_config\_\*() API should not be used as it is deprecated. The uvm\_resource\_db API should not be used due to quirks in its behavior.

### **6.2 Rule: Do create configuration classes to hold configuration values.**

To provide configuration information to agents or other parts of the testbench, a configuration class should be created which contains the bits, strings, integers, enums, virtual interface handles, etc. that are needed. Each agent should have its own configuration class that contains every piece of configuration information used by any part of the agent. Using the configuration class makes it convenient and efficient to use a single uvm\_config\_db #(config\_class)::set() call and is type-safe. It also allows for easy extension and modification if required.

### **6.3 Guideline: Don't use the configuration space for frequent communication between components.**

Using the resource database for frequent communication between components is an expensive way to communicate. The component that is supposed to receive new configuration information would have to poll the configuration space which would waste time. Instead, standard TLM communication should be used to communicate frequent information changes between components. TLM communication does not consume any simulation time other than when a transaction is being sent.

Infrequent communication such as providing a handle to a register model is perfectly acceptable.

### **6.4 Rule: Pass virtual interface handles from the top level testbench module into the testbench by using the uvm\_config\_db API.**

Since the uvm\_config\_db API allows for any type to be stored in the UVM resource database, this should be used for passing the virtual interface handle from the top level testbench module into the UVM test. This call should be of the form uvm\_config\_db #(virtual bus\_interface)::set(null, "uvm\_test\_top", "bus\_interface", bus\_interface); Notice the first argument is null which means the scope is uvm\_top. The second argument now lets us limit the scope of who under uvm\_top can access this interface. We are limiting the scope to be the top level uvm\_test as it should pull the virtual interface handles out of the resource database and then add them to the individual agent configuration objects as needed.

## Coverage

### **7.1 Guideline: Place covergroups within wrapper classes extended from uvm\_object.**

Covergroups are not objects or classes. They can not be extended from and also can't be programmatically created and destroyed on their own. However, if a covergroup is wrapped within a class, then the testbench can decide at run time whether to construct the coverage wrapper class.

Please refer to the SystemVerilog Guidelines for more on general covergroup rules/guidelines.

## End of Test

### **8.1 Rule: Do use the phase objection mechanism to end tests. Do use the phase\_ready\_to\_end() function to extend time when needed.**

To control when the test finishes use the objection mechanism in UVM. Each UVM phase has an argument passed into it (phase) of type uvm\_phase. Objections should be raised and dropped on this phase argument. For more information, visit the End of Test article.

### **8.2 Rule: Only raise and lower objections in a test. Do not raise and lower objections on a transaction by transaction basis.**

In most cases, objections should only be raised and lowered in one of the time consuming phases in a test. This is because the test is the main controller of what is going to happen in the testbench and it therefore knows when all of the stimulus has been created and processed. If more time is needed in a component, then use the phase\_ready\_to\_end() function to allow for more time. See Phasing/Transactors for more information.

Because there is overhead involved with raising and dropping an objection, Mentor Graphics recommends against raising and lowering objections in a driver or monitor as this will cause simulation slowdown due to the overhead involved.

## Callbacks

### **9.1 Guideline: Do not use callbacks.**

UVM provides a mechanism to register callbacks for specific objects. This mechanism should not be used as there are many convoluted steps that are required to register and enable the callbacks. Additionally callbacks have a non-negligible memory and performance footprint in addition to potential ordering issues. Instead use standard object oriented programming (OOP) practices where callback functionality is needed. One OOP option would be to extend the class that you want to change and then use the UVM factory to override which object is created. Another OOP option would be create a child object within the parent object which gets some functionality delegated to it. To control which functionality is used, a configuration setting could be used or a factory override could be used.

## MCOW (Manual Copy on Write)

**10.1 Guideline: If a transaction is going to be modified by a component, the component should make a copy of it before sending it on.**

The general policy for TLM 1.0 (blocking put/get ports, analysis\_ports, etc.) in SV is MCOW ( "moocow" ). This stands for manual copy on write. Once a handle has passed across any TLM port or export, it cannot be modified. If you want to modify it, you must manually take a copy and write to that. In C++ and other languages, this isn't a problem because a 'const' attribute would be used on the item and the client code could not modify the item unless it was copied. SV does not provide this safety feature, so it can be very dangerous.

Sequences and TLM 2.0 connections don't follow the same semantic as TLM 1.0 connections and therefore this rule doesn't apply.

## Command Line Processor

**11.1 Rule: Don't prefix user defined plusargs with "uvm\_" or "UVM\_".**

User defined plusargs are reserved by usage by the UVM committee and future expansion. For more information, see the UVM/CommandLineProcessor article.

**11.2 Guideline: Do use a company and/or group prefix to prevent namespace overlap when defining user defined plusargs.**

This allows for easier sharing of IP between groups and potentially companies as it is less likely to have a collision with a plusarg name. For more information, see the UVM/CommandLineProcessor article.

---

# Appendix - Glossary of Terms

---

## Doc/Glossary

---

This page is an index to the glossary of various terms defined and used in the Cookbook. Each glossary term has its own page which contains links to other related Cookbook pages. A summary is below:

Glossary Index:

### A

#### **Advanced Verification Methodology (AVM)**

A verification methodology and base class library written in SystemVerilog and created by Mentor Graphics in 2006, the AVM was the precursor to OVM and UVM. It provided a framework for component hierarchy and TLM communication to provide a standardized use model for SystemVerilog verification environments. AVM is not recommended for new projects, refer instead to Open Verification Methodology (OVM)

#### **Agent**

An Agent is a verification component for a specific logical interface. The logical interface can be implemented as a SystemVerilog interface - as a collection of wires. An agent's job is to drive activity on the interface, or monitor activity on the interface, or both. It normally contains driver, monitor, sequencer, coverage and configuration functionality. Methodology base class libraries typically provide a base class for an agent as part of a component framework, although they are normally vacuous.

#### **Analysis**

Functionality in a Verification Environment which involves taking input from one or more DUT interface monitors in the form of abstracted transactions and analyses of the contents, for correctness or in order to record coverage statistics.

See also predictor, score board, analysis port, monitor.

#### **Analysis Port**

In UVM, an API for reporting abstract transactions to interested parties in the verification environment. The API is used to connect a producer component (e.g. a monitor) to one or more consumer components, also called subscribers (these are typically analysis components, e.g. scoreboards or coverage objects). Connection is a simple one-time activity. Transactions are then written to the port and the API automatically calls each subscriber's write() API in turn, with that transaction handle. The process takes zero simulation time and has lowest possible overhead.

#### **Aspect Oriented Programming (AOP)**

A software programming methodology which provides a framework and syntax for separation of concerns, across a hierarchy of objects. Unlike traditional Object Oriented Programming (OOP), Aspect Oriented Programming is not restricted to separation of concerns using hierarchy alone. It allows orthogonal slices of functionality to be defined separately and then stitched together with core functionality and other aspects and compile/elaboration time. Individual methods of a class can be augmented (prepended or appended) or replaced by a contribution from an aspect. It is often used to add a layer of debug functionality, or to specify test variants in a concise manner. As a powerful language feature, it comes with a challenge, to avoid overuse or avoid adding complexity that is hard to debug or maintain at a

later date. Various proponents and opponents can be found. Notable languages incorporating some degree of AOP functionality are 'e' and 'vera'. SystemVerilog has no AOP features.

### **Assertion**

A software language construct which evaluates a condition and causes a fail or error message if the condition is false. A concise way to perform a check, often a check which is a prerequisite to subsequent processing, e.g. a check for null pointer before the pointer is used. In high-level Design/Verification Language terms, an assertion is often used to refer to a property/assertion construct which checks for a condition (the property) using a concise language which allows interrogation of signal states, transitions, in some order or within some time constraints. The properties referred to in assertion statements can be built up from sequences and can become considerably complex; they can also be used with some formal tools for static evaluation. SystemVerilog Assertions (SVA) is a part of the SystemVerilog design/verification language and contains a powerful set of syntax to make these condition checks. Other assertion languages include PSL (property specification language).

### **Assertion Coverage**

A kind of Functional Coverage which measures which assertions have been triggered. SystemVerilog contains syntax to add coverage to individual properties specified using SVA. Such coverage is useful to know whether the assertion is coded correctly, and whether the test suite is capable of causing the condition that is being checked to occur.

## **B**

### **Black Box**

A style of design verification where the design under test is considered opaque. The design can only be stimulated and observed using the available port connections, with no internal activity or signaling available for analysis.

### **Bus Functional Model (BFM)**

A model which represents the behavior of a protocol (e.g. an addressable bus) at the signal function level. It is capable of providing the required signal activity to implement the features of the bus, under control of some API or higher level abstraction.

## **C**

### **Class**

In Object-Oriented Programming (OOP), a software (or HVL) language construct which encapsulates data and the methods which operate on that data together in one software definition. The definition can typically inherit base functionality from a parent class and optionally augment or override that functionality to create a more specific definition. Classes can be instantiated and constructed one or more times to create objects.

See also: [Module](#)

### **Clock Domain Crossing**

A clock domain crossing is when a signal moves from the output of a flip-flop clocked by one clock (and therefore part of that clock domain) to the input of a flip-flop clocked by a different clock (a second clock domain). At the boundary (crossing) between these domains, some synchronization is normally required, the nature of that depending on the relationship between the two clocks. Verification techniques, methodology and tools exist to perform sanity checks on this particular area of DUT functionality.

### **Code Coverage**

The capability for a simulator to record statistics on execution of each line of HDL code in a DUT, as a rough approximation for the quality of the verification. Several variants exist which track more detailed coverage, e.g. of multiple logical paths through a design depending on conditionals. Common variants include branch, statement, toggle, expression, condition and fsm coverage.

### **Component**

A software entity which performs some functionality and can be connected to other components. In a class-based verification environment such as UVM, a component is a pseudo-static object, created in a preset way using the UVM API, is part of the testbench hierarchy, and exists for the lifetime of the simulation.

### **Configuration Object**

An abstraction of one or more configuration values of various types, used within a verification environment to make shared persistent data available to one or more components, for the purposes of configuring those components to perform their function in a particular way, or to equip them with some higher level pseudo-static state of the DUT. More dynamic data is expressed in transaction objects - configuration objects are for less-frequently-changing or constant state.

### **Configuration Space**

A set of state values that can be explored during verification, representing the set of possible configuration values for a DUT. Configuration can refer to soft configuration (e.g. programmed configuration register values or initialization pins) or hard parameterization. It is most often used to refer to soft config.

### **Constrained Random**

A verification methodology where test stimulus is randomized in the HVL before being applied, rather than specified in a directed manner. The randomization is tailored (constrained) to a sensible subset of all possible values, to narrow the set of all possible stimulus down to a subset of legal, useful, interesting stimulus. Constrained random can also be applied to other aspects of the test including DUT configuration selection and testbench topology. SystemVerilog has constructs built in to the language to support constrained random testing.

### **Constraint**

In Constrained Random Stimulus, an item which restricts a particular aspect of a stimulus transaction (normally one transaction class member) to be a preset value, or bound within a specified range of values, or having a particular relationship to the value of another member of the randomized class. A set of constraints can be built up to narrow the random space for a stimulus transaction down to a legal, useful or interesting subset. During randomization, constraints are evaluated until a value random result is created that satisfies all constraints, or an error is produced if this is not possible.

### **Consumer**

In software, an entity that receives data from a producer. The job of the consumer is to consume data or information. That information was created by a producer. In HVL, the data is normally an abstract transaction object. The abstract transaction object represents information - for example a READ transaction might contain an address and a data. That transaction is produced by the producer, and delivered to the consumer. This kind of transaction communication is referred to as transaction level communication. In UVM, there is an API to provide TLM (transaction level modeling) communication and Analysis Ports. The TLM standard and analysis ports facilitate consistent producer/consumer connectivity.

### **Coverage**

A statistical measurement of verification progress made by observing language features, including Code Coverage and Functional Coverage.

### **Coverage Class**

In Functional Coverage methodology, a class (e.g. written in SystemVerilog) which contains coverage constructs to measure a particular reported entity (e.g. a transaction from a monitor, or a completed check from a scoreboard). Contained within a class in order to keep the coverage definition separate from the machinery of the test environment / interface monitors

### **Coverage Closure**

The process in verification of successive refinement of the stimulus that is given to the DUT and the checks that are performed, following the Verification Plan, until a sufficient measurement of coverage is reached w.r.t. the set of coverage goals implied by the Verification Plan. When coverage closure is reached, the DUT is considered verified (according to the criteria of the plan as executed)

## **D**

### **Device Under Test (DUT)**

The piece of code (RTL, gate level, System C models, etc.) which is being exercised by a testbench.

### **Driver**

A verification component responsible for taking transactions written at a particular level of abstraction and converting them to a lower-level of abstraction, according to the protocol being verified. Most drivers have as their 'lower level abstraction' the bus functional model which causes individual signal wires on the interface to be updated and reacted to. A typical driver would accept a transaction and convert it to signal changes.

### **Dual Top**

An architectural arrangement where the testbench is contained within one toplevel Verilog model, and the DUT is contained within another. Often associated with the requirements of Emulation, where portions of the test environment are required to be synthesizable.

## **E**

### **E Reuse Methodology (eRM)**

A methodology for use of the Verisity 'e' language and the 'specman' tool to create verification components and test environments following a consistent architecture. Some of its features, such as sequences, are reused in the SystemVerilog OVM and UVM methodologies.

### **Emulation**

The acceleration of verification from what is possible in simulator software to a hardware platform where the simulation is synthesized into a netlist of logic running on real hardware (e.g. an FPGA implementation of logic cells and programmable interconnect). Trades off a more complex compilation step for vastly faster runtime cycle-per-second performance. Emulation also can constrain the ability to perform debug on the running test environment. Good emulation solutions workaround those tradeoffs to provide a full-featured verification environment for productive verification of large designs or complex state spaces, or system-level evaluation of real firmware running on emulated hardware.

## F

### **Fork-join**

A Verilog construct which splits a thread of activity into one or more new subthreads, each executing independently on the simulator, and subsequently 'joined' together in a predefined way. SystemVerilog enhances the syntax available for specification of how to join threads, so that the coder can choose to wait for all threads to complete (the default) [fork/join], or for the first one to complete [fork/join\_any], or just to move on leaving them running independently [fork/join\_none].

### **Functional Coverage**

A kind of coverage supported in language syntax and measured in verification tools to reflect the desire expressed in a Verification Plan to apply particular test stimulus and checks to a DUT in order to fully verify it. Items in a Verification Plan can map to named/grouped functional coverage points which are defined in the language. SystemVerilog code can trigger those measurements by evaluating a specified triggering condition, and can report a data value which is recorded in a preset manner, for subsequent reporting. Functional Coverage is a key part of today's constrained random methodology, where the test intent is specified in the set of functional coverage points defined and measured in the context of random stimulus application, not in a set of manually directed tests with their own stimulus and checks.

## G

### **Golden Model**

A model which represents some existing known good behavior of a function, normally created outside the scope of the verification activity. When one exists, it is useful in a verification environment to form part of a scoreboard / predictor or other checker arrangement to enable self-checking.

### **Graph Based Verification**

One of a number of 'intelligent testbench automation' features which optimizes the constrained random stimulus space using a graph representing interesting paths through the detailed possibilities of the protocol, in order to provide faster coverage closure compared to normal randomization of stimulus.

## H

### **Hierarchy**

An arrangement of entities which by their definition may contain zero or more sub-entities, each of which are also hierarchical. In the design/verification activity, can be used in relation to design modules (Verilog modules can contain instances of other modules) and of testbench component hierarchy (SystemVerilog UVM component objects can instantiate child component objects) or of sequence stimulus hierarchy (sequences can invoke sub-sequences)

## I

### **InFact**

Mentor Graphics Questa InFact is the industry's most advanced testbench automation solution. Using Graph Based Verification techniques, it targets as much functionality as traditional constrained random testing, but achieves coverage goals 10X to 100X faster. This enables engineering teams to complete their functional verification process in less time, and/or to expand their coverage goals, testing functionality that previously fell below the cut line. Questa InFact also generates tests that an engineer might not envision, reaching difficult corner cases that alternative testing techniques

typically miss, and also has the built-in functionality of generating a covergroup based on the stimulus class.

### **Interface**

In SystemVerilog, a construct representing a bundle of defined wires, normally used to represent the signals which comprise a single instance of a protocol interface between a DUT and a testbench. A useful abstraction used during testbench definition and hookup, avoiding the need to replicate declarations for each member signal along the way. Can be provided to a class-based testbench via a virtual interface

### **Interrupt**

In CPU hardware/software, a system event which is conveyed by hardware to the CPU where it causes normal software execution to be suspended and a designated subroutine to be executed in order to service the interrupt, clear it, and return to the normal execution thread. Hierarchy and priority normally exist, allowing interrupts to be interrupted in turn, if required. For Verification, it is desirable to emulate this behavior by allowing a sequence of stimulus to be interrupted by some waited-upon event and a special sequence substituted and executed on the driver representing the interrupt service routine and the set of reads and writes that are required there, before returning to the existing running sequence. UVM sequencer/driver architecture allows this kind of override, priority, hierarchy with various APIs for locking / grabbing / setting arbitration priority.

## **M**

### **Matlab**

A language and programming tool / simulation environment for modeling algorithms and datapaths at a higher level of abstraction. It also provides graphical visualization of these higher level mathematical functions.

### **Metrics**

Measurable statistics, in the case of a verification environment, representing progress, performance and completeness of the verification activity. Notable metrics are code coverage, functional coverage, bug rates, source code complexity, project-to-project reuse statistics, and project management measurements

### **Model**

An abstracted software representation of a hardware or software function, containing some approximation of the set of external connections and some approximation of the internal behavior. The intent of a model could be to document the specification of a function to be implemented/measured, or to serve as part of a checking environment for a design which implements the function, to be compared with the model's execution of any given stimulus, or to deliver an alternate level of detail of the function for some optimized usage.

### **Module**

A named, instantiatable design unit in Verilog - a static collection of behavioral or RTL or gate-level functionality with a predefined set of inputs and outputs, which can be formed into a hierarchy or can enclose submodules - instances of other modules in a hierarchy. Unlike Classes, Modules are static - they are created at elaboration time and therefore exist at time 0 before the simulation starts, and remain in a fixed structure for the entire simulation.

### **Monitor**

A component responsible for watching pins wiggle on a bus and converting those pin wiggles back into transactions. The transactions are then sent out through an analysis\_port. The analysis\_port may be connected to a scoreboard, a predictor, a coverage class, etc. The monitor doesn't do checking directly.

## O

### **Open Verification Methodology (OVM)**

A base class library written in SystemVerilog language, implementing verification environment and stimulus features such as component hierarchy, abstract transaction connections (TLM), sequences of stimulus transactions, message reporting, testbench construction phasing, and incorporating OOP techniques such as configuration patterns and factory pattern substitution. First Released in 2008, and developed as a joint effort between Mentor Graphics and Cadence Design Systems, building on Mentor's existing SystemVerilog AVM architecture and incorporating patterns and use models from Cadence's 'e' reuse methodology (eRM). OVM formed the basis for the Universal Verification Methodology (UVM) in 2010. OVM is portable code known to run on all SystemVerilog-compliant simulators and is in widespread use today.

## P

### **Package**

In SystemVerilog, a software entity which provides an encapsulation of a set of software artifacts (types, variables, class definitions) within a single namespace. This serves to protect symbols from collision with those in other package namespaces, and also to provide a mechanism for import of a set of functionality into another software entity for reuse.

### **Parameterized Class**

As discussed in Parameters a class may have elaboration-time parameters which allow variation of its runtime representation, those parameters containing values - integral or string or class handle, or types, e.g. class types. The feature is similar to the templates in C++ language. Once parameters are added to a class, it is no longer possible to instantiate the class without specifying those parameters (although defaults can be provided), and any hierarchy above the object of that class type needs to pass down the required parameter values which are ultimately set to concrete values at a high level of the hierarchy. In verification methodology such as UVM, parameterized classes are heavily used to implement TLM connections, factory patterns and other methodology features. They are advisable only for verification component hierarchy usage, in situations where the DUT and its interfaces are parameterized and the test environment has to be capable of testing more than one variant of those. However, they are not recommended for use on data/transaction/sequence or analysis component classes, where more abstract forms of representation at runtime can replace them.

### **Parameterized Tests**

The application of Parameters and Parameterized Classes to the problem of providing a library of tests with variations in applicability, perhaps selectable at runtime, or perhaps aligned with the parameters required for the DUT instance to be tested. The test class typically instantiates the environment(s) which in turn instantiate Agent(s) containing driver(s) or monitor(s). For methodologies which pass a virtual interface to drivers and monitors to connect to the actual DUT, any parameterization in the DUT (and its interfaces) needs passed down to the drivers/monitors, via the Agents, Environments, and ultimately the tests.

### **Parameters**

In Verilog and SystemVerilog, the ability to specify pseudo-constants on module or class declarations, which are resolved to actual values during the elaboration phase of compilation, for the purpose of making those entities more reusable and flexible to more than one situation. Different instances of a module or objects of a class type can have different parameter values. parameter values can be passed down hierarchy to sub-modules or instances of types within a class. Parameterized modules and classes introduce an aspect to software construction that provides flexibility but adds a

complexity and maintenance burden, ideally this powerful feature is used sparingly. Often soft run-time configuration variables can suffice.

### **Phasing**

Phasing is a stepwise approach to the construction of a verification environment at runtime and the execution of required stimulus and the completion of the test when stimulus is done. UVM has an API which enables components to participate in this step by step process in a predetermined and familiar manner. The construction of a structured test environment with TLM connections is done in a predetermined manner to enable smart hierarchy and connectivity management, and at runtime there are phases which can map to different kinds of stimulus or can be user-defined as required for the application. Most verification environments use the simplest possible subset of the available phases: build, connect, run.

### **Pipelined**

An attribute of a hardware design of a logic function or connectivity protocol. Pipelining is the storage of intermediate representations of progress within the execution of a logic function or protocol transfer in an overlapping form, allowing a transform which takes more than one clock cycle to complete and overlap with others, in order to improve overall throughput (but not normally to affect or impact end to end latency). The depth of the pipeline is the amount of overlap that is possible in a particular design, depending on the nature and amount of the intermediate storage elements, and constrained by the protocol itself, in particular any feedback loops or possible stalls in execution. Verification of pipelined logic or protocols requires stimulus to match that complexity - simple back-to-back stimulus is insufficient. Techniques can be used in Drivers and Monitors to support pipelining.

### **Power Architecture Coverage**

A kind of Functional Coverage which is specific to the verification of low-power domain architectures, often referring to a standard convention such as Unified Power Format (UPF) which breaks down the possible states and features in a predictable manner. Functional verification of normal DUT traffic and features must coexist with verification of power architecture and the various state transitions that can occur, and their interactions with normal stimulus.

### **Predictor**

A verification component which acts alongside or within a scoreboard or checker, comparing two or more observed behaviors on the design under test for correctness. The predictor may transform an observed behavior into another form which can be more directly compared for consistency. That transformation may involve reference to a Doc/Glossary/Golden Model which may implement functional transformation, aggregation or disaggregation, or more complex topologies of input versus output.

### **Producer**

In software, an entity that delivers data to a Consumer. The job of the producer is to produce data or information. That information will be passed to a consumer. In HVL, the data is normally an abstract transaction object. The abstract transaction object represents information - for example a READ transaction might contain an address and a data. That transaction is produced by the producer, and delivered to the consumer. This kind of transaction communication is referred to as transaction level communication. In UVM, there is an API to provide TLM (transaction level modeling) communication and Analysis Ports. The TLM standard and analysis ports facilitate consistent producer/consumer connectivity.

### **Program Block**

Program blocks in SystemVerilog exist in order to mimic the scheduling semantics that a PLI application has interacting with a Verilog simulator. As such they are not normally required in an OVM/UVM testbench in SystemVerilog where there is direct access to the DUT pins from the language with no PLI requirement. Mentor Graphics does not

recommended their use.

## Q

### **Questa Verification IP (QVIP)**

The verification components in the Questa Verification IP library fit any verification environment. By facilitating and enhancing the application of transaction-level modeling (TLM), advanced SystemVerilog testbench features (such as constrained random testing), modern verification methodologies (such as OVM and UVM), and seamlessly integrating with other Mentor tools (such as Questa Verification Management, Questa Questa inFact and Veloce), Questa Verification IP increases productivity even further. Questa Verification IP delivers a common interface across the library of protocols. This results in a scalable verification solution for popular protocols and standard interfaces, including stimulus generation, reference checking, and coverage measurements that can be used for RTL, TLM, and system-level verification. Verification with Questa Verification IP is straight forward: simply instantiate it as a component in your testbench. The built-in capabilities of Questa Verification IP automatically provides the entries for the coverage database so you have the metrics in place to track whether all necessary scenarios are covered. Questa Verification IP is also integrated with the Questa debug environment and makes use of the transaction viewing capability in Questa so that you can get both signal-level and transaction-level debugging capabilities.

## R

### **Register Model**

A kind of Model which represents the behavior of a set of named addressable registers, grouped into hierarchies as required, divided into named fields with attributes and specified behaviors, all representing the normal kind of behavior of a bus-addressable configuration/control/status/data interface. The structures occur so frequently in typical designs, and conform to a small set of common feature patterns, so are ideal for usage of a common modeling solution. One such model solution is found in the UVM. It enables abstract reads/writes to named registers or fields, via a collaborating bus driver, and modeling of current or expected values which can be predicted and checked, in collaboration with a bus monitor. It can be used as a configuration object elsewhere in the testbench where decisions have to be made about stimulus delivery, reassembly, or checking, based on known soft configuration values stored in register fields. It can be used to inform coverage objects which seek to measure design coverage related to different register settings.

### **Regression**

The running of a suite of tests repeatedly in some cycle, in order to determine forward progress of a design/verification activity, and catch any unexpected backwards progress (regression) so that debug may determine whether a design bug or a verification bug exists, and what needs to be done to move that test to a passing state and continue with forward progress. A regression suite has various attributes - it may require one or more verification environments / testbenches, it may use parameters to alter design or testbench configuration, there may be test stimulus which is directed, or constrained random, or graph-based, and there may be different knobs controlling checks to be performed, amount of debug data to collect, and extent of simulation. Underlying machinery is used to distribute the compilation and running of the various tests on simulators and other tools, often requiring parallel run distribution across multiple compute environments. Underlying machinery can control the intent of the regression (what stimulus to run) and capture the results of that run and/or cumulative runs on order to present data to the verification resources to aid their productivity or their ability to achieve Coverage Closure.

### **Resource**

In UVM, a configuration variable or object, held within a Resource Database, for global access by both components who subscribe to configuration of that type, and those who are responsible for specifying the configured value.

### **Responder**

A verification component which, similar to a driver, interacts with a lower level of abstraction such as the individual signals on a protocol interface, in order to participate in a protocol. Unlike a driver, responders do not initiate stimulus traffic from a sequence, they respond (slave-like) to traffic observed on the interface in order to maintain legal protocol. Their execution may be controlled by configuration (specifying how they are to respond in general) including some constrained randomization possible within the allowed configuration space, or by sequences of configuration changes (altering the way they respond in a controlled way), or by maintaining some state, e.g. a memory model or register model, which reflects persistent data values deposited by earlier traffic and subsequently retrieved accurately by the responder.

## **S**

### **Scoreboard**

A component responsible for checking actual and expected values. The actual and expected values will be delivered via analysis\_port/export connections to the scoreboard from monitors and/or predictors. The scoreboard may also record statistical information and report that information back at the end of a simulation.

### **Sequence**

A class-based representation of one or more stimulus items (Sequence Items) which are executed on a driver. Can collaborate in a hierarchy for successive abstraction of stimulus and can participate in constrained random setup to enable highly variable sets of stimulus above the randomization possible in an individual transaction. Sequences can represent temporal succession of stimulus, or parallel tracks of competing or independent stimulus on more than one interface. They can be built up into comprehensive stress test stimulus or real world stimulus particular to the needs of the protocol. UVM has comprehensive support the automated definition and application of sequences. In its simplest form, a sequence is a function call (a functor), which may request permission to communicate with a driver using a sequence item. This complicated sounding interaction is not so complicated. A sequence asks for permission to send a transaction (sequence item) to the driver. Once it has been granted permission by the sequencer, then the transaction is passed to the driver.

### **Sequence Item**

A class-based abstract transaction representing the lowest level of stimulus passed from a sequence to a driver. Also known as a Transaction.

### **Sequencer**

A component responsible for co-ordinating the execution of stimulus in the form of sequences and sequence items from a parent sequence, ultimately feeding a driver component with transactions. UVM/OVM provide a standard sequencer component with preset arbitration and locking methods for complex sequence stimulus. At its simplest, a sequencer can be thought of as a fancy arbiter. It arbitrates who gets access to the driver, which represents who gets access to the interface.

### **Shmoo**

A Test/Verification technique where test stimulus is deliberately applied with a time offset from other test stimulus in order to force a rendezvous inside the design under test of stimulus arrival or state change, combined with a sweep of relative timing offset to explore simultaneous arrival in addition to a plus-or-minus offset between two or more stimulus

events. The sweep distance should equal or exceed the known pipeline depth of the logic under test in order to flush out pipeline definition bugs. Unlikely to be achieved by normal randomization alone, more usually an orchestrated or semi-directed test sequence calling sub-sequences in a time-controlled manner.

### **Subscriber**

In a verification environment, a component which is a consumer of transaction data via an analysis port and associated connection made during verification environment construction. A UVM base class is supplied for the simplest case of subscriber, and other classes and macros are available to build more complex subscribers which feed from more than one analysis port traffic. Most analysis components are subscribers in this way.

## **T**

### **Test**

A class-based representation of a verification scenario. A test consists of a verification environment constructed in a particular way, some particular stimulus run upon it in a particular way, checks, coverage, and debug capabilities enabled or configured according to intent of the engineer or regression environment. The toplevel of a component hierarchy in UVM.

### **Test Plan**

Part of a Verification Plan, a document or documents which capture the detailed intent of the verification team for completing coverage closure on the design under test. The Test Plan is typically in spreadsheet or other structured form and contains detailed testbench requirements and links to functional coverage, assertions or specific tests. The Test Plan is sometimes referred to as 'verification plan' or 'requirements spreadsheet' or 'trace matrix' or 'coverage spreadsheet'.

### **Transaction**

An abstraction of design activity on a signal interface into a class-based environment, to allow higher-order verification intent to be expressed more concisely. The unit of data that is passed along TLM connections in UVM. When the stimulus strategy involves UVM sequences, the transaction is also known as a Sequence Item. A transaction is a collection of information that is passed from point A to point B, and may have a beginning time and an ending time. A transaction may have relationships with other transactions - like transaction X is the parent of transaction Y (and transaction Y is the child of transaction X).

### **Transactor**

A generic term for any verification component that actively interacts with the DUT interfaces (e.g. a driver or responder)

### **Two kingdoms**

This is one way of describing a class based technique that allows transactors to talk to a DUT. The technique relies on an concrete implementation of an abstract interface in an interface and a handle to the abstract class in the transactor. It used to workaround some of the complexities associated with virtual interfaces.

## **U**

### **UVM Verification Component (UVC)**

A set of software and related artifacts providing a self-contained, plug'n'play verification solution for a particular protocol interface or logic function, compatible with the UVM. A UVC consists of one or more configurable Agents with a set of familiar APIs and ports, defining the underlying signal [Doc/Glossary/InterfaceInterface and TLM Sequence Item, a Sequence Library of example stimulus and protocol compliance test stimulus, a comprehensive Protocol

Coverage Model, some example Analysis components and tests, and documentation including a reference protocol Verification Plan. The resulting set of collateral provides a coherent, complete solution for protocol or function compliance testing. The term 'Verification Component' is sometimes used interchangeably with 'Verification IP (VIP)'.

### **Unified Coverage Database (UCDB)**

A database developed by Mentor Graphics for capture of code coverage, functional coverage, and other metrics together in one place for post-regression analysis and reporting. Forms the basis of the UCIS industry standard in development in Accellera.

### **Unified Power Format (UPF)**

A format for specifying the design intent for power control features in a design such as power domains, power isolation, power state transitions and register/memory retention regions. Verification of the power design intent can be made using formal and simulation tools, based on the content of a UPF description. The UPF format is defined by IEEE 1801-2009.

### **Universal Verification Methodology (UVM)**

A standard API and reference base class library implementation in SystemVerilog describing a standardized methodology for verification environment architecture, to facilitate rapid, standardized verification environment creation and stimulus application/coverage closure on a design under test. Can be viewed as an extension to the SystemVerilog language to avoid the necessity to reinvent the wheel on common architectural traits such as component hierarchy build-out, transaction level modeling connections, higher-order stimulus application, common configuration and substitutions, register modeling, and reporting. Derived from the Open Verification Methodology (OVM), and created in 2010 by Mentor Graphics, Cadence and Synopsys working in the context of the Accellera committee. Other Accellera participants have since contributed to the implementation and its maintenance.

## **V**

### **Veloce**

Mentor Graphic's emulation platform for high-performance simulation acceleration and in-circuit emulation of complex integrated circuits.

### **Verification**

The process of evaluating a programmatic representation of a hardware design for correctness according to its specifications, including the use of techniques such as simulation, mathematical and formal evaluation, hardware emulation, application of directed tests or random stimulus in the context of a programmatic verification environment.

### **Verification Architecture Document (VAD)**

A high level verification planning document that records all of the verification planning, strategy and decisions. This is a 'parent' document and all other verification documents are typically derived from it. It may contain coverage information, or that may be split out into a separate Verification Implementation Document (VID)

### **Verification Implementation Document (VID)**

A verification document containing more detailed implementation planning for example a coverage plan, highlighting all the locations where coverage is measured, and associated conventions for coverage naming and configuration. Often part of or a sub-document of a Verification Architecture Document (VAD)

### **Verification Methodology Manual (VMM)**

A standard API and reference base class library implementation in SystemVerilog describing a standardized methodology for verification environment architectures. It was created by Synopsys and was derived from the Vera

RVM (Reference Verification Methodology). Still supported by Synopsys although superseded by the Universal Verification Methodology (UVM).

### **Verification Plan**

A document or documents which capture the intent of the verification team for completing coverage closure on the design under test. It is good practice to split the document into separate verification architecture and optional implementation documents, and a separate Test Plan in spreadsheet or other structured form for detailed requirements and links to functional coverage.

### **Virtual Interface**

In SystemVerilog, a virtual interface is a handle variable that contains a reference to a static interface instance. Virtual interfaces allow the class-based portion of a verification environment to connect to physical interfaces containing signal definitions in the static module hierarchy.

### **Virtual Sequence**

A sequence which controls stimulus generation across more than one sequencer, co-ordinate the stimulus across different interfaces and the interactions between them. Usually the top level of the sequence hierarchy. AKA 'master sequence' or 'co-ordinator sequence'. Virtual sequences do not need their own sequencer, as they do not link directly to drivers. When they have one it is called a virtual sequencer.

### **Virtual Sequencer**

A virtual sequencer is a sequencer that is not connected to a driver itself, but contains handles for sequencers in the testbench hierarchy. It is an optional component for running of virtual sequences - optional because they need no driver hookup, instead calling other sequences which run on real sequencers.

## **W**

### **White Box**

A style of design verification where the design under test is considered semi-transparent. In addition to stimulating and observing the Doc/Glossary/Black Box ports on its periphery, its internal signals or elements of interest such as state machines or FIFOs can also be stimulated and observed by the testbench.

### **Worker Sequence**

A sequence which calls lower level API sequences to build up compound activities such as dut configuration, loading a memory, initializing a feature. Usually a worker sequence would only be sending eventual sequence items to a single sequencer (as opposed to a virtual sequence which sends to multiple sequencers) A worker sequence is a fancy name for a sequence that does work by calling other sequences. Collections of worker sequences can be built up, one upon the other. In the software world this might be called chains of function calls. Each function calling lower level functions in turn. Worker sequences call (or start, or send) lower level sequences. A worker sequence may hide details of sequences - like calling start. See Sequences/Hierarchy for an example.

**For the latest product information, call us or visit: [www.mentor.com](http://www.mentor.com)**

© 2012-13 Mentor Graphics Corporation, all rights reserved. This document contains information that is proprietary to Mentor Graphics Corporation and may be duplicated in whole or in part by the original recipient for internal business purposes only, provided that this entire notice appears in all copies. In accepting this document, the recipient agrees to make every reasonable effort to prevent unauthorized use of this information. All trademarks mentioned are the trademarks of their respective owners.

Corporate Headquarters	Silicon Valley	Europe	Pacific Rim	Japan
<b>Mentor Graphics Corporation</b>	<b>Mentor Graphics Corporation</b>	<b>Mentor Graphics</b>	<b>Mentor Graphics (Taiwan)</b>	<b>Mentor Graphics Japan Co., Ltd.</b>
8005 SW Boeckman Road	46871 Bayside Parkway	Deutschland GmbH	Room 1001, 10F	Gotenyama Garden
Wilsonville, OR 97070-7777	Fremont, California 94538 USA	Arnulfstrasse 201	International Trade Building	7-35, Kita-Shinagawa 4-chome
Phone: 503.685.7000	Phone: 510.354.7400	80634 Munich	No. 333, Section 1, Keelung Road	Shinagawa-Ku, Tokyo 140-0001
Fax: 503.685.1204	Fax: 510.354.1501	Germany	Taipei, Taiwan, ROC	Japan
<b>Sales and Product Information</b>	<b>North American Support Center</b>	Phone: +49.89.57096.0	Phone: 886.2.87252000	Phone: +81.3.5488.3030
Phone: 800.547.3000	Phone: 800.547.4303	Fax: +49.89.57096.400	Fax: 886.2.27576027	Fax: +81.3.5488.3021