Universal Verification Methodology
UVM Cookbook

COOKBOOK

FUNCTIONAL VERIFICATION

www.verificationacademy.com

Contents

Articles

Introduction **1**

Cookbook Introduction 1

Cookbook Overview Diagram 2

Acknowledgements 3

UVM Basics **4**

UVM Basics 4

Testbench Basics 4

UVM Components 6

The UVM Factory 8

Phasing 11

UVM Driver 15

UVM Monitor 16

UVM Agent 19

UVM Sequences 25

UVM Sequence Items 27

UVM Configuration Database (uvm_config_db) 28

Using Packages 31

Testbench Architecture **34**

Testbench Architecture 34

Building a UVM Testbench 36

Sequencer-Driver Connections|Connecting the Sequencer and Driver 45

Block-Level Testbench 47

Integration-Level Testbench 58

Dual Top Architecture 71

DUT-Testbench Connections **74**

Connecting the Testbench to the DUT 74

Interfaces and Virtual Interfaces 75

The UVM Configuration Database (uvm_config_db) 76

Virtual Interface BFMs 78

Handling Parameterization 84

Abstract-Concrete Class Connections 90

Parameterized Tests 93

Configuring a Test Environment **96**

Testbench Configuration 96
Configuring Sequences 100
Using a Parameter Package 103
Accessing Configuration Resources from a Sequence 108
Macro Cost-Benefit Analysis 110

Analysis Components & Techniques **112**

Analysis Components and Techniques 112
Analysis Port 113
Analysis Connections 117
Predictors 123
Scoreboards 124
Metric Analyzers 131
Post-Run Phases 133

End Of Test Mechanisms **136**

End-of-Test and Objection Mechanisms 136
Objections 138

Sequences **141**

UVM Sequences 141
Sequence Items 143
Transaction Methods 144
Sequence API 149
Sequencer 153
Driver-Sequence API 155
Generating Stimulus with UVM Sequences 162
Overriding Sequences and Sequence Items 170
Virtual Sequences 172
Virtual Sequencers (Not Recommended) 179
Hierarchical Sequences 184
The Sequence Library 188
Sequence-Driver Use Models 192
Unidirectional Protocols 193
Bidirectional Protocols 196

Pipelined Protocols 201
Arbitrating Between Sequences 212
Sequence Priority 218
Locking or Grabbing a Sequencer 219
Slave Sequences (Responders) 224
Wait for a Signal 230
Interrupt Sequences 235
Stopping a Sequence 241
Layering Sequences 242

The UVM Messaging System **247**

UVM Messaging 247
Using Messaging 249
UVM Report Catcher 253
Testing Message Status 255
Command-Line Verbosity Control 256
Messaging in Sequences 256

Other Stimulus Techniques **258**

CBasedStimulus 258

Register Abstraction Layer **271**

The UVM Register Package 271
Register Model Overview 276
Register Model Structure 286
Complex Address Maps 295
Specifying Registers 299
The Register Layer Adapter 301
Integrating a UVM Register Model in a Testbench - Overview 304
Integrating a UVM Register Model in a Testbench - Implementation 309
"Quirky" Registers 313
Register Model Coverage 317
Backdoor Accesses 322
Generating Register Models 325
Register-Level Stimulus 326
Memory-Level Stimulus 336
Register Sequence Examples 341
Built-in Register Sequences 347

Configuring Registers 352

Register-Level Scoreboards 354

Register-Level Functional Coverage 360

Testbench Acceleration through Co-Emulation **365**

Emulation 365

Separate Top-Level Modules 367

Split Transactors 372

Back Pointers 377

Defining an API 381

Example 384

Example Driver 393

Example Agent 397

Example Top-Level Model 403

Debug of SV and UVM **407**

UVM Debugging 407

Built-In Debug 407

Using Verbosity for Debug 418

Command-Line Processor 423

UVM Connect - SV-SystemC interoperability **427**

UVM Connect(UVMC) 427

UVMC Connections 429

UVMC Conversion Layer 431

UVMC Command API 434

UVM Versions and Compatibility **438**

UVM1.2/Summary 438

UVM1800.2/Summary 442

Appendix - Deployment **446**

UVC/UVMVerificationComponent 446

Package/Organization 456

Questa/CompilingUVM 462

Appendix - Coding Guidelines **464**

SV/Guidelines 464
SV/PerformanceGuidelines 475
UVM/Guidelines 495
UVM/PerformanceGuidelines 502

Appendix - Glossary of Terms **527**

Doc/Glossary 527

Article Licenses

License 546

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 Online Methodology Cookbook, whose recipes can be adapted and applied in many different ways by our field experts, customers, and partners alike.

As the UVM has continued to be refined in Accellera, we have updated the *UVM Cookbook* accordingly. This latest update was prompted by the adoption of UVM as IEEE 1800.2 in November, 2017, with the subsequent release by Accellera of a compatible Reference Implementation library shortly thereafter.

The other significant change in the industry since UVM was first introduced is the incredible increase in the size and complexity of designs. What used to be considered a "system" back then is now considered just one piece of a much larger system today. This has greatly increased the demand for faster execution platforms on which to perform verification, and as such we are seeing more and more users adopting emulation, hardware acceleration and FPGA prototyping as part of their functional verification flows. The requirement to preserve their UVM infrastructure throughout a project has led us to update our recommendation for how to organize a UVM testbench, particularly at the level at which the UVM testbench interacts with the Device under Test (DUT), which you will see throughout this book.

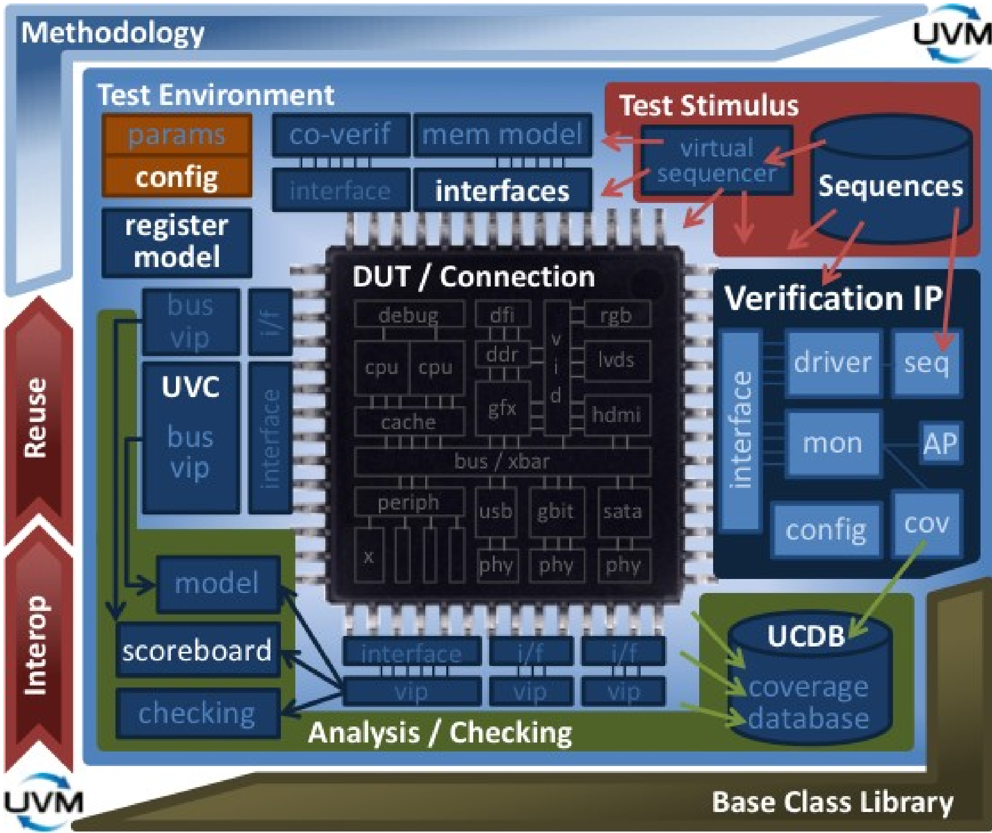
We encourage you to share this valuable resource with your friends and colleagues. One of the greatest advantages of publishing this book online is the ability of readers like yourself to provide immediate detailed feedback and to give us the ability to respond quickly without having to wait for another hardcopy edition to be published. We hope this Cookbook will assist you in incorporating the UVM into your particular application and generally improving your functional verification process.

Find us online at http://verificationacademy.com/cookbook

Cookbook Overview Diagram

The UVM Cookbook is now part of Mentor's Verification Academy.

Visit us at http://verificationacademy.com/cookbook



Acknowledgements

UVM Cookbook Authors:

2017-2018 Cookbook Upgrade for Dual Domain Emulation-Friendly Architecture and IEEE 1800.2 Standard Compliance

- Michael Horn
- Hans van der Schoot
- Mark Peryer
- Tom Fitzpatrick
- John Stickley

We would like especially to thank our Beta reviewers for this version of the UVM Cookbook: David Lacey, Courtney Fricano, Jason Sprott, Ian Perryman, Jeff Kellam, Neil Johnson, Ashish Amonkar, Masoud Madani, Ravi Kalyanaraman and Erik Jessen.

Original Cookbook

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

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, Hager Fathy, Jennifer Adams, John Carroll, John Amouroux, Jason Polychronopoulos, John Stickley, Nigel Elliot, Peet James, Ray Salemi, Shashi Bhutada, Tim Corcoran, Jon Craft, and Tom Fitzpatrick.

UVM Basics

UVM Basics

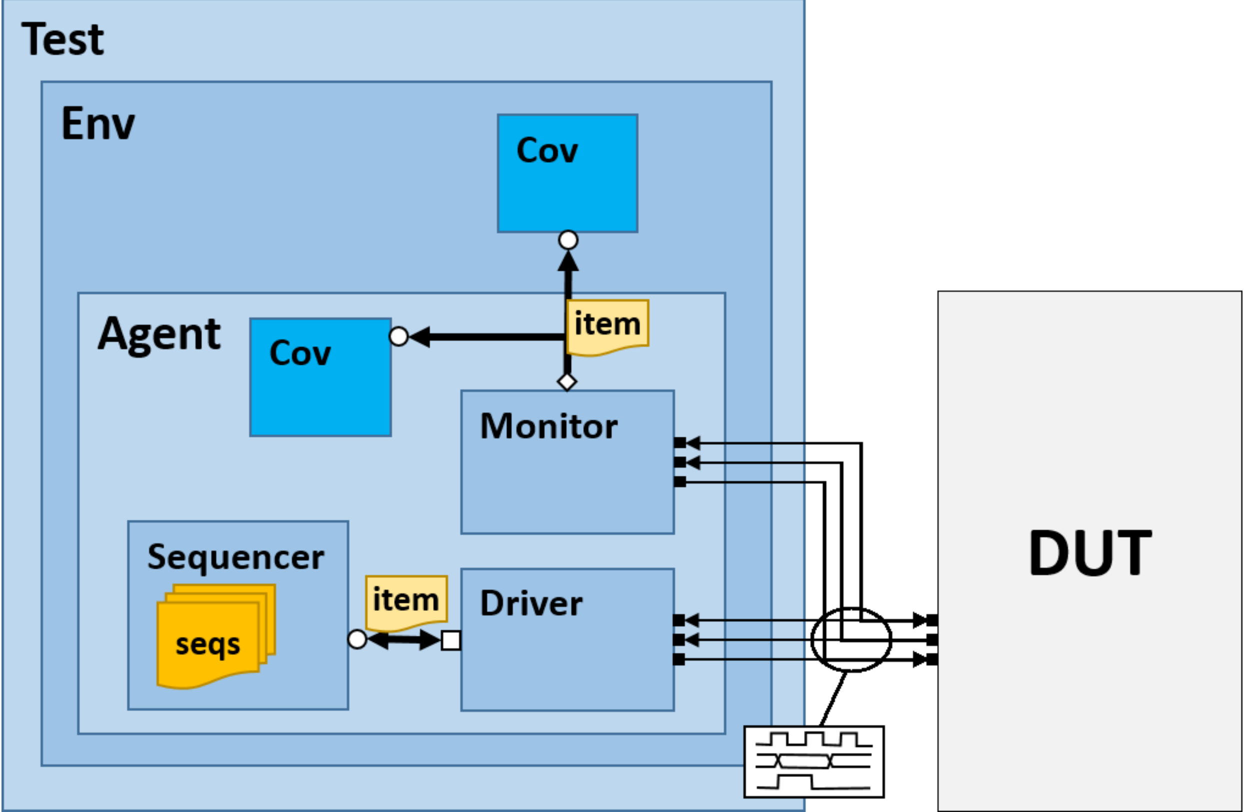
Before we can get into discussing the recipes presented in the UVM Cookbook, we have to make sure that we're all talking about the same ingredients. This chapter introduces the UVM concepts that the reader should know in order to understand the recipes presented herein. This section will be incredibly valuable to new UVM users, but experienced UVM users may be able to just straight to the UVM Testbench chapter.

Testbench Basics

UVM Testbench Basics

The UVM employs a layered, object-oriented approach to testbench development that allows “separation of concerns” among the various team members. Each component in a UVM testbench has a specific purpose and a well-defined interface to the rest of the testbench to enhance productivity and facilitate reuse. When these components are assembled into a testbench, the result is a modular reusable verification environment that allows the test writer to think at the transaction level, focusing on the functionality that must be verified, while the testbench architect focuses on how the test interacts with the Design Under Test (DUT).

The Design Under Test (DUT) is connected to a layer of transactors (drivers, monitors, responders). These transactors communicate with the DUT at the pin level by driving and sampling DUT signals, and with the rest of the UVM testbench by passing transaction objects. They convert data between pins and transactions, i.e. from/to signal to/from transaction level. The testbench layer above the transactor layer consists of components that interact exclusively at the transaction level, such as scoreboards, coverage collectors, stimulus generators, etc. All structural elements in a UVM testbench are extended from the uvm_component base class.



The lowest level of a UVM testbench is interface-specific. For each interface, the UVM provides a uvm_agent that includes the driver, monitor, stimulus generator (sequencer) and (optionally) a coverage collector. The Agent thus embodies all of the protocol-specific communication with the DUT. The Agent(s) and other design-specific components are encapsulated in a *uvm_env* Environment component which is in turn instantiated and customize by a top-level *uvm_test* component.

The uvm_sequence_item – sometimes referred to as a transaction – is a *uvm_object* that contains the data fields necessary to implement the protocol and communicate with the DUT. The uvm_driver is responsible for converting the sequence_item(s) into "pin wiggles" on the signal-level interface to send and receive data to/from the DUT. The sequence_items are provided by one or more uvm_sequence objects that define stimulus at the transaction level and execute on the agent's *uvm_sequencer* component. The sequencer is responsible for executing the sequences, arbitrating between them and routing sequence items between the driver and the sequence.

The uvm_monitor is responsible for passively observing the pin-level behavior on the DUT interface, converting it into sequence items and providing those sequence items to analysis components in the agent or elsewhere in the testbench such as coverage collectors or scoreboards. UVM Agents also have a *configuration object* that allows the test writer to control aspects of the agent as the testbench is assembled and executed.

By providing a uniform interface to the testbench, a UVM Agent isolates the testbench and the UVM Sequence from details of the interface implementation. A sequence that provides data packets, for example, can be reused with different UVM Agents that may implement AHB, PCI or other protocols. A UVM testbench will typically have one agent per DUT interface.

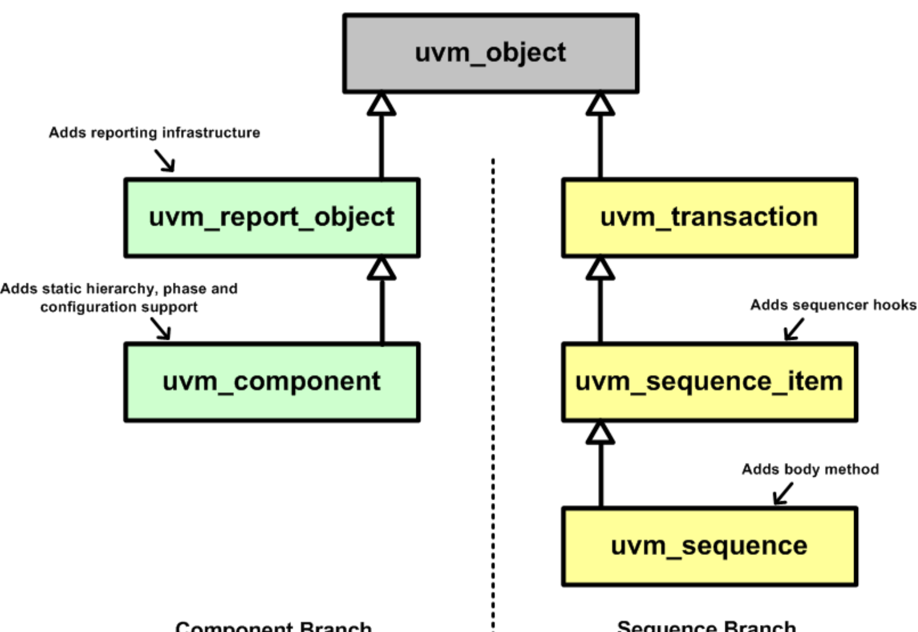
For a given design, the UVM Agents and other components are encapsulated in a uvm_env environment component, which is typically design-specific. Like an agent, an environment typically has a configuration object associated with it that allows the test to control aspects of the environment as well as to control the agents instantiated in the environment. Because environments are themselves UVM components, they can be assembled into a higher-level environment. As block-level designs are assembled into subsystems and systems, the block-level UVM environment associated with the block may be reused as a component in the subsystem-level environment, which can itself be reused in the system-level testbench.

Once the environment has been defined, the uvm_test will instantiate, configure and build the environment, including customizing key aspects of the overall testbench, including ..*choosing variations of components to be used in the environment ..*choosing UVM Sequences to be run either in the background or as the main portion of the test ..*defining configuration objects for the environment, sub-environment(s) (if any) and agent(s) in the testbench

The UVM test is started from an initial block in the top-level HVL module by calling *run_test()*.

UVM Components

A UVM testbench is composed of component objects extended from the uvm_component base class. When a uvm_component derived class object is created, it becomes part of the testbench hierarchy which persists for the duration of the simulation. This contrasts with the sequence branch of the UVM class hierarchy which involves transient objects - objects that are created, used and destroyed (i.e. garbage collected) once dereferenced.



The (quasi) static uvm_component hierarchy is used by the UVM reporting infrastructure to print the scope of a component issuing a report message, by the configuration process to determine which components can access a configuration object, and by the UVM factory to apply factory overrides. This component hierarchy is represented by a linked list built up incrementally as each component is created. The hierarchical location of each component is determined by the name and parent arguments passed to its create method at the time of construction.

For instance, in the code fragment below, an apb_agent component is created within the spi_env. Assuming the spi_env is instantiated in the top-level test component with the name "m_env," the hierarchical path name of the agent is the concatenation of the spi_env component's name, "uvm_test_top.m_env", the "dot" (".") operator, and the name passed as the first argument to the "create()" method, resulting in a hierarchical name for the agent of "uvm_test_top.m_env.m_apb_agent". Any references to the agent would need to use this string name.

```
//
// 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 "uvm_test_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 class which lies at the heart of 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 to be implemented by the user as required. A phase level virtual method that is not implemented results in the component effectively not participating in that phase.

Also embedded into the uvm_component base class is support for a configuration table to store configuration objects that are relevant for the component's child nodes in the testbench hierarchy. When the uvm_config_db API is used, this static hierarchy is employed as part of the path mechanism to control which components may access a given 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 a UVM component is created during the build phase, the factory is used to construct the component object. The UVM factory enables a component to be swapped for another of a compatible, derived type using a factory override. This is a useful technique for altering the functionality of a testbench without changing the testbench source code directly, which would require recompilation and hinder reuse. There are a number of coding conventions required for the factory to work and these are outlined in the article on the UVM Factory.

The UVM package contains a number of extensions (i.e. derived classes) of the uvm_component base class for common testbench components. Most of these extensions are very "thin", i.e. they are literally just a small extension of the uvm_component class to add a new name space. While these are non-critical and in principle a uvm_component class could be used instead still, they do help with "self-documentation" as they indicate clearly the

kind of component that is actually represented, such as a driver or monitor. Additionally, there are analysis utilities available that also use these extraneous base classes as clues to help establish a picture of the testbench hierarchy. On the other hand, some of the pre-built uvm_component extensions are in fact building blocks providing more profound added value, by instantiate concrete sub-components. The following table summarizes the available UVM component classes directly derived from the uvm_component base class:

Class	Description	Contains sub-components?
uvm_driver	Encapsulates sub-components for sequence communication with the uvm_sequencer	No
uvm_sequencer	Encapsulates sub-components for sequence communication with the uvm_driver	No
uvm_subscriber	Encapsulates a uvm_analysis_export and associated virtual *write* method to implement analysis transaction processing	No
uvm_env	Basis for aggregating verification components around a DUT, or other envs in case of vertical (sub-)system integration	Yes
uvm_test	Basis for a concrete top level test	Yes
uvm_monitor	Basis for a concrete monitor transactor	Yes
uvm_scoreboard	Basis for a concrete scoreboard	Yes
uvm_agent	Basis for concrete agent including a sequencer-driver pair and a monitor	Yes

The UVM Factory

The UVM Factory

The purpose of the UVM factory is to enable an object of one type to be substituted with an object of a derived type without changing the testbench structure or even the testbench code. The mechanism used is referred to as an override, by either instance or type. This functionality is very handy for changing sequence behavior or replacing one version of a component by another. Any two components to be swapped must be polymorphically compatible. This includes the requirement that all the same TLM interface handles and TLM objects must be created by the replacement component. Additionally, in order to take advantage of the factory certain coding conventions must be followed.

Factory Coding Convention 1: Registration

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

- A 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 parameterized 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;

// Parameterized component factory registration macro
`uvm_component_param_utils(this_t)

// For a class derived from an object (i.e. uvm_object,
uvm_transaction, uvm_sequence_item, uvm_sequence etc.)
class my_item extends uvm_sequence_item;

`uvm_object_utils(my_item)

// For a parameterized object class
class my_item #(int ADD_WIDTH=20, int DATA_WIDHT=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 with a prototype template that must be adhered to by users. 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 can be 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 handle to the class to its declaration handle in the testbench after the name and parent arguments are 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 using the create method. 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
```

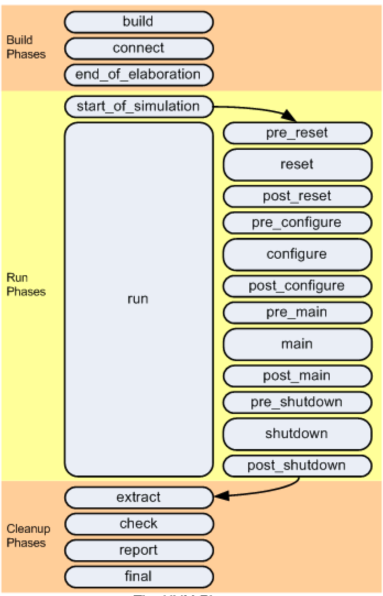
Phasing

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.



The UVM Phases

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_phase() tasks are executed in parallel. Transactors such as drivers and monitors will nearly always use this phase.

Parallel Run-Time Phases

NOTE: The following run-time phases execute in-order, in parallel with the run_phase phase. These phases should only be called from the test and the env to start sequences. Drivers, monitors and other components should not implement these phases.

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 behavior. 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.

UVM Driver

Overview

The UVM driver is responsible for communicating at the transaction level with the sequence via TLM communication with the sequencer and converting between the sequence_item on the transaction side and pin-level activity in communicating with the DUT via a virtual interface. As the name implies, drivers typically get a sequence_item and use that information to drive signals to the DUT and may, in certain applications, also receive a pin-level response from the DUT and convert it back into a sequence_item for the sequence to complete the transaction. A driver may also function as a "responder" (i.e. in "slave mode") in which the driver reacts to pin-level activity in the interface to communicate with a sequence that then sends a response transaction back to the driver to complete the protocol transaction. When its agent is configured to be in passive mode, the driver is, by definition, not instantiated in the agent.

Anatomy of a UVM Driver

A user-defined driver component is a proxy class derived from a *uvm_driver* base class and contains a BFM which is a SystemVerilog interface. The *uvm_driver* base class provides a seq_item_port that gets connected by the agent to the seq_item_export of the agent's *uvm_sequencer*. Usually, responses are passed back to the sequence through the seq_item_port as well, but certain applications may require that responses be sent back to the sequencer via the *rsp_port* of the driver. For a detailed discussion of connecting a driver to a sequencer, please see Sequencer-Driver Connections.

UVM Driver API

The *uvm_driver* is designed ultimately to interact with a *uvm_sequence* running on the connected *uvm_sequencer*. The details of this API are described here.

UVM 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 behavior of the driver.

There are a large number of potential stimulus generation use models for the sequence driver combination, most of which are discussed here.

UVM Monitor

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 are composed of a proxy class which should extend from uvm_monitor and a BFM which is a SystemVerilog interface. The proxy should have one analysis port and a virtual interface handle that points to a BFM interface.

```
class wb_bus_monitor extends uvm_monitor;
`uvm_component_utils(wb_bus_monitor)

  uvm_analysis_port #(wb_txn) wb_mon_ap;
  virtual wb_bus_monitor_bfm m_bfm;  //BFM handle
  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 );
    wb_mon_ap = new("wb_mon_ap", this);
    m_config = wb_config::get_config(this); // get config object
    m_bfm = m_config.WB_mon_bfm; // set local virtual if property
    m_bfm.proxy = this; //Set BFM proxy handle
  endfunction

  task run_phase(uvm_phase phase);
    m_bfm.run();  //Don't start the BFM until we get to the run_phase
  endtask

  function void notify_transaction(wb_txn item);  //Used by BFM to
return transactions
    wb_mon_ap.write(item);
  endfunction : notify_transaction

endclass
```

```
interface wb_bus_monitor_bfm (wishbone_bus_syscon_if wb_bus_if);

  import wishbone_pkg::*;

  //------------------------------------------
  // Data Members
  //------------------------------------------
  wb_bus_monitor proxy;


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

  task run();
    wb_txn txn;

    forever @ (posedge wb_bus_if.clk)
      //Capture protocol pin activity into txn
      proxy.notify_transaction(txn);
    end
  endtask

endinterface
```

Passive Monitoring

As in a scientific experiment 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. Additionally, the BFM will not being observing data until instructed to do so by the proxy to ensure that the BFM is aligned with UVM phasing.

Recognizing Protocols

A monitor must have knowledge of a protocol in order to detect recognizable patterns in signal activity. Detection can be done by writing protocol-specific state machine code in the monitor BFM's run() task. This code waits for a pattern of key signal activity by observing through the pin 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.

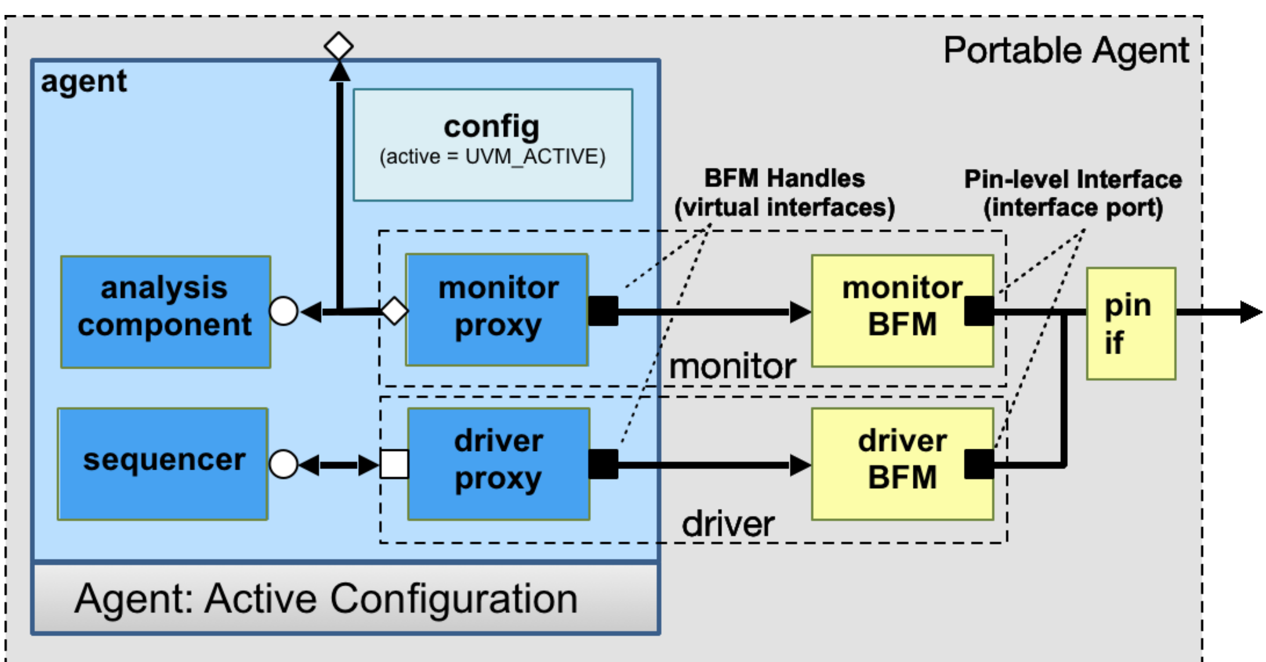
Example Monitor

```
//Full run task from monitor BFM
task run();
  wb_txn txn;

  forever @ (posedge wb_bus_if.clk)
    if(wb_bus_if.s_cyc) begin // Is there a valid wb cycle?
      txn = wb_txn::type_id::create("txn"); // create a new wb_txn
      txn.adr = wb_bus_if.s_addr; // get address
      txn.count = 1;  // set count to one read or write
      if(wb_bus_if.s_we)  begin // is it a write?
        txn.data[0] = wb_bus_if.s_wdata;  // get data
        txn.txn_type = WRITE; // set op type
        while (!(wb_bus_if.s_ack[0] |
wb_bus_if.s_ack[1]|wb_bus_if.s_ack[2]))
          @ (posedge 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
          wb_bus_if.s_stb[0]:  begin
            while (!(wb_bus_if.s_ack[0])) @ (posedge wb_bus_if.clk); //
 wait for ack
              txn.data[0] = wb_bus_if.s_rdata[0];  // get data
            end
```

```
          wb_bus_if.s_stb[1]:  begin
              while (!(wb_bus_if.s_ack[1])) @ (posedge wb_bus_if.clk);
// wait for ack
              txn.data[0] = wb_bus_if.s_rdata[1];  // get data
            end
        endcase
      end // else: !if(wb_bus_if.s_we)
      proxy.notify_transaction(txn);
    end
  endtask
```

UVM Agent

A UVM agent is a verification component "kit" for a given logical interface such as APB or USB. The agent includes a SystemVerilog interface encapsulating the corresponding set of interface signals, two SystemVerilog interfaces representing the monitor and driver BFMs, and a SystemVerilog package including the various classes that make up the overall agent class component. The agent class itself is a container class for a sequencer, a driver proxy and a monitor proxy, plus any other verification components deemed relevant such as a functional coverage collector or a scoreboard. Proxies are simply class objects providing the same API as "normal" class objects. The driver proxy and monitor proxy communicate with the rest of a UVM testbench in the usual way, while also accessing respectively the driver and monitor BFM interface via a virtual interface handle. A complete monitor is thus composed of the monitor proxy and monitor BFM working together as a pair, and the same is true for a driver [1]. The agent also has an analysis port that is connected to the analysis port of the monitor, enabling a user to connect external analysis components to the agent without needing to know its internal structure. The agent is the lowest level hierarchical block in a testbench and its exact structure depends on its configuration, which can vary from one test to another per the agent configuration object. The classes and interfaces together constitute a portable, or reusable Agent.



Let's examine how an APB agent is composed, configured, built, and connected. The APB agent's pin interface, apb_if, is coded in the file apb_if.sv. The monitor BFM interface named apb_monitor_bfm has an apb_if port. The

driver BFM named apb_driver_bfm also has an apb_if port. The BFMs define tasks and functions to interact with the signals in the apb_if pin interface. The driver and monitor proxies do not directly access the pins, which is to be kept local to the BFMs. The file apb_agent_pkg.sv contains the SystemVerilog package with the various class files for the APB agent. Any component using files from this package, like an env, imports this package.

```
package apb_agent_pkg;

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

`include "apb_seq_item.svh"
`include "apb_agent_config.svh"
`include "apb_driver.svh"
`include "apb_coverage_monitor.svh"
`include "apb_monitor.svh"
typedef uvm_sequencer#(apb_seq_item) apb_sequencer;
`include "apb_agent.svh"

//Reg Adapter for UVM Register Model
`include "reg2apb_adapter.svh"

// Utility Sequences
`include "apb_seq.svh"
`include "apb_read_seq.svh"
`include "apb_write_seq.svh"

endpackage: apb_agent_pkg
```

Note that the apb_sequencer type is actually a typedef that simply parameterizes the uvm_sequencer base component with the sequence_item type being used.

The Agent Configuration Object

The agent has a configuration object which defines:

- The topology of the agent's sub-components (determines what gets constructed)
- The handles for the BFM virtual interfaces used by the driver and monitor proxies
- The behavior of the agent

By convention, UVM agent configuration classes have a data member of type uvm_active_passive_enum that defines whether the agent is active (UVM_ACTIVE) with the sequencer and driver proxy constructed, or passive (UVM_PASSIVE) with neither the driver proxy nor sequencer constructed. This parameter is called active and by default set to UVM_ACTIVE.

Whether other sub-components are built or not is controlled by additional configuration data members which should have descriptive names. For instance, if there is a functional coverage collector, there ought to be a bit that controls whether this coverage collector is built or not, suitably named has_functional_coverage.

The configuration object contains handles for the BFM virtual interfaces used by the driver proxy and the monitor proxy. The configuration object is constructed and configured in the test and it is at this top level where the virtual interface handles are assigned to the virtual interfaces passed in from the testbench module.

The agent configuration object may also contain other data members to control how the agent is configured or behaves. For instance, the configuration object for the APB agent has data members to set up the memory map and determine the APB PSEL lines that are activated with associated address map.

The configuration class should have all configuration data members default to common values.

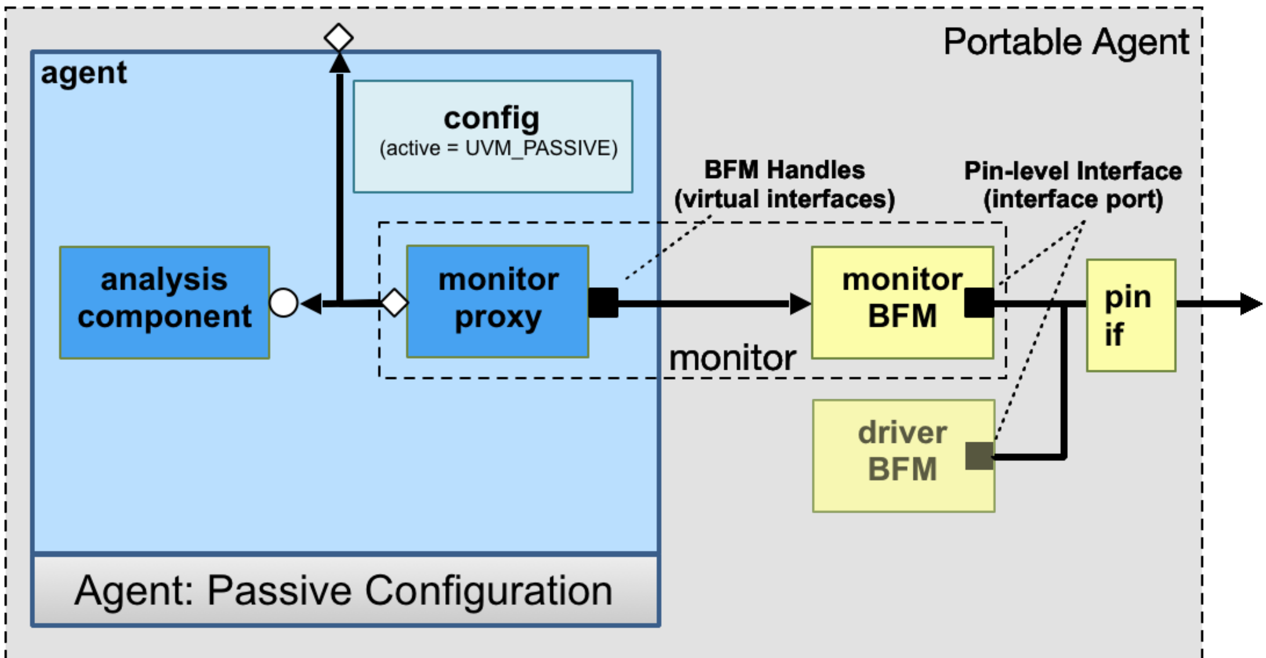
The following code example shows the configuration object for the APB agent.

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

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

// BFM Virtual Interfaces virtual
apb_monitor_bfm mon_bfm; virtual
apb_driver_bfm                    drv_bfm;

//------------------------------------------
// Data Members
//------------------------------------------
// Is the agent active or passive
uvm_active_passive_enum active = UVM_ACTIVE;
// Include the APB functional coverage collector
bit has_functional_coverage = 0;
// Include the APB RAM based scoreboard
bit has_scoreboard = 0;
//
// Address decode for the select lines:
// Address decode for the select lines:
int no_select_lines = 1;
int apb_index = 0; // Which PSEL is the monitor connected to
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 an agent's build phase are determined by the contents of its configuration object. The first action is to get a handle to the configuration object. Then, when sub-components are to be constructed, the values of pertinent configuration fields determine whether they should be constructed or not.



The exceptions to the rule are the monitor proxy and monitor BFM which are always present since they are used irrespective of whether the agent is active or passive. The driver proxy is only built if the agent is in active mode. The driver BFM requires additional thought and depending on reuse objectives, different actions may be taken as follows.

Driver BFM Instantiation

As a SystemVerilog interface, the driver BFM is instantiated and created at static elaboration time. Therefore the BFM state machines should generally not start automatically (i.e. self-start) in order to prevent signals being driven or samples before the RTL and/or the rest of the testbench is ready. Instead, a start indication from the corresponding proxy should be used as trigger for a BFM to start its activity. This allows the BFM to be synchronized to the standard UVM Phasing mechanism and also keeps a driver silent if the agent is configured as passive. In addition to being silent, the driver must also not drive when in passive mode. This means that either the driver must default to being in a tri-state ('hz) mode for all outputs or the driver must not be instantiated when in passive mode.

To decide whether or not to instantiate the driver, consider the level of agent reuse desired. For simple block-to-top reuse where RTL code replaces driver functionality, the driver is not needed in any circumstances and should not be instantiated. If there is a possibility that the driver BFM may yet need to drive signals, then instantiation is required as is the utilization of tri-state signal values. To control instantiation, a parameter could be used with a generate statement.

The Agent Connect Phase

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

- Monitor's analysis port to agent's analysis port [1]
- Sequencer's seq_item_pull_export to driver's seq_item_pull_port (for active agent only)
- Analysis sub-component's analysis export to monitor's analysis port (where analysis sub-components exist)
- Monitor/driver's proxy virtual interface to configuration object's monitor/driver BFM virtual interface [2]

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 proxy and monitor proxy 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 (m_cfg == null)
    if( !uvm_config_db #( apb_agent_config )::get(this, "",
"apb_agent_config",m_cfg) ) `uvm_fatal(...)
  // 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 phase);
  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);
  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:

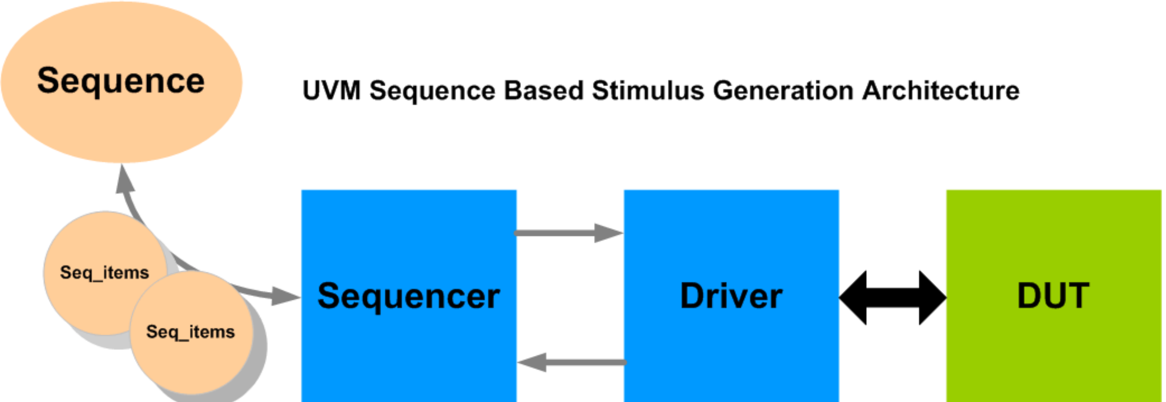
References

[1] https://verificationacademy.com/patterns-library/implementation-patterns/environment-patterns/bfm-proxy-pair-pattern

UVM Sequences

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 can 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 hiearchy stays in place for the life-time of the simulation.

Simplified UVM Inheritance Diagram

uvm_object

Adds reporting infrastructure

uvm_report_object

uvm_transaction

Adds static hierarchy, phase and configuration support

uvm_component

Adds sequencer hooks

uvm_sequence_item

Adds body method

uvm_sequence

Component Branch

Sequence Branch

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.

UVM Sequence Items

The UVM stimulus generation process is based on sequences controlling the behavior 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.

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.

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 behavior 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 optimizes 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.

UVM Configuration Database (uvm_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 two or more components or objects. Use uvm_config_db::set to put information into the database and use uvm_config_db::get to retrieve information from the database. The uvm_config_db class is a type-parameterized class, and consequently the database behaves as if it were partitioned into many type-specific "mini databases." There are no limitations on the type parameter, which can be a class, a uvm_object, a built-in type like a bit, byte, or a virtual interface, etc.

There are two typical uses of the uvm_config_db. The first is to pass virtual interfaces from the HDL/DUT domain to the test, and the second is to pass configuration objects down through the testbench hierarchy.

The set method

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

- **T** is the type of the resource, or element, being added - usually a virtual interface or a configuration object.
- **cntxt** and **inst_name** together form a scope that is used to locate the resource within the database; it is formed by appending the instance name to the full hierarchical name of the context, i.e. {cntxt.get_full_name(),".",inst_name}.
- **field_name** is the name given to the resource.
- **value** is the actual value or reference that is put into the database.

An example of putting virtual interfaces into the UVM configuration database is as follows:

```
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:

- Use **"uvm_test_top"** because it is more precise and hence more efficient than "*"; it works in general except when your top level test component does something other than super.new(name , parent) in its constructor, in which case the instance name must be adjusted accordingly.
- Use **"null"** in the first argument as this code is in a top level module rather than a uvm_component.
- Use the parameterized uvm_config_db::set() method instead of the set_config_[int,string,object]() methods which are deprecated in UVM1.2. See UVM1.2 Summary for more details.

An example of configuring agents inside an env is as follows:

```
class env extends uvm_env;

  ahb_agent_config m_ahb_agent_config;

  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" , m_ahb_agent_config );
    ...
  endfunction
endclass
```

This code sets the configuration for the AHB agent and all its child components. Two things to note:

- Use **"this"** as the first argument to ensure that only *this* agent's configuration is set, and not of any other ahb_agent in the component hierarchy.
- Use **"m_ahb_agent*"** to ensure that both the agent and its children are in the look-up scope. Without the '*' only the agent itself would be, and its driver, sequencer and monitor sub-components would be unable to access the configuration.

The get method

The full signature of the get method is **bit uvm_config_db #(type T = int)::get(uvm_component cntxt , string inst_name , string field_name , ref T value);**

- **T** is the type of the resource, or element, being retrieved - usually a virtual interface or a configuration object.
- **cntxt** and **inst_name** together form a scope that is used to locate the resource within the database; it is formed by appending the instance name to the full hierarchical name of the context, i.e. {cntxt.get_full_name(),".",inst_name}.
- **field_name** is the name given to the resource.
- **value** holds the actual value or reference that is retrieved from the database; the get() call **returns** 1 if that retrieval succeeds, or else 0 indicating that no resource of this type and with this context and name exists in the database.

An example of getting virtual interfaces from the configuration database is as follows:

```
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 assign it into the correct agent's configuration object. A meaningful error message is provided when the database lookup fails.

An example of retrieving configuration for a transactor is as follows:

```
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
```

Again, a few notes are in order:

- Use **"this"** as the context argument.
- Use **""** as the instance name.
- Use the return value of the get() call to check whether it fails and a useful error message should be given.
- Use the parameterized uvm_config_db::get() method instead of the get_config_[int,string,object]() methods which are deprecated in UVM1.2. See UVM1.2 Summary for more details.

Precedence Rules

Two sets of precedence rules apply to uvm_config_db. Firstly, during the build phase a set() call in a context higher up the component hierarchy takes precedence over a set() call occurring lower down the hierarchy. Secondly, in case of identical contexts or after the build phase, the last set() call takes precedence over the earlier one. For more details on these rules, please consult the UVM reference manual or look directly at the verbosity messages from the UVM code.

Using 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.

A SystemVerilog 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". Packages should be used for developing UVM testbenches to collect and organize the various class definitions developed to implement agents, envs, sequence libraries, test libraries and so forth.

UVM Package Coding Guidelines

Naming a package and package file

A package should be named with a _pkg suffix. The name of the file containing the package should reflect the name of the package and have a .sv extension. As an 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 suffix denotes that the file contains a package. Both of these conventions are useful to humans and machine parsing scripts.

Including classes in a package

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, i.e. there should be no further `include statements inside the `included files themselves.

Justification: Declaring classes in separate individual files makes them easier to maintain, and also conveys the package content more clearly.

Importing packages and package elements

The contents of a package may reference contents of other packages as enabled by package imports. Such external package imports should be declared at the top of the package as the first statements of the package "body". Individual files like class templates that may be included should not do separate imports.

Justification: Localizing all package imports in one place makes the package dependencies clear. Placing package imports in other parts of the package or inside `included files is less visible and more likely to cause ordering issues and potential type clashes.

Organizing package files into a directory

All files included in a given package should be put 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: A single include directory for package files facilitates compilation flow set-up, and also aids reuse since all the files for a package can be gathered together easily.

Below is an example of a package file for a UVM env. This env contains two agents (SPI and APB) and a register model, which 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

Users are often confused by the fact that the SystemVerilog package defines a scope. This means that everything declared within a package, along with the elements of other packages imported into the package are visible only within the scope of that package. If a package is imported into another scope (i.e. another package or a module), only the elements of the imported package are visible and not the elements of any packages imported by the imported package itself. Hence, if elements of these other packages are required in the new scope, 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
```

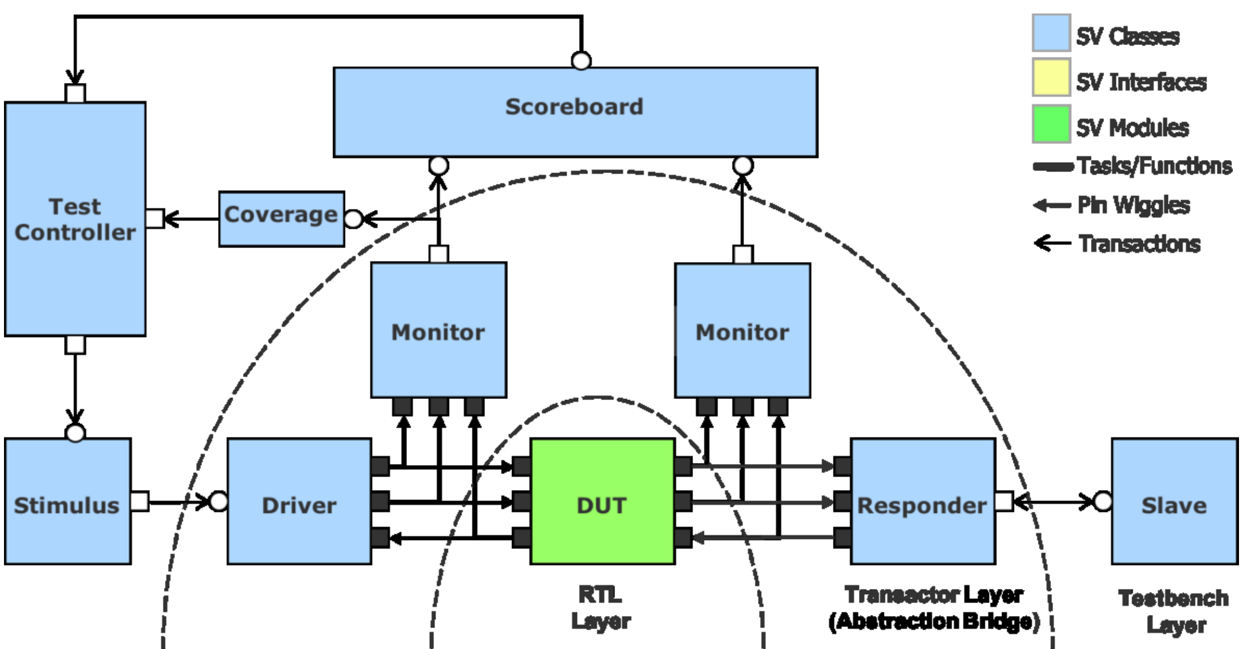
```
// visible within the current spi_test_pkg scope
import spi_env_pkg::*;

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

endpackage: spi_test_pkg
```

Testbench Architecture

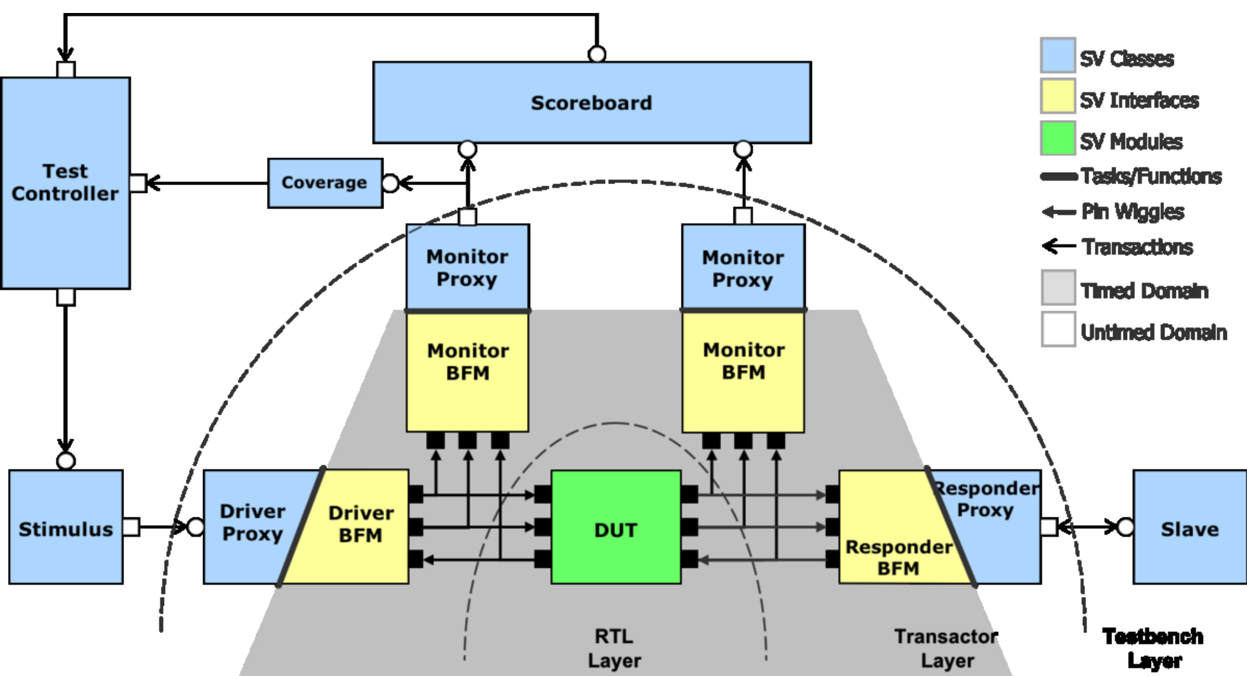
Testbench Architecture

UVM Testbench Architecture [1]

A UVM testbench is built using SystemVerilog (dynamic) class objects interacting with SystemVerilog (static) interfaces and modules in a structured hierarchy. The hierarchy is composed of layers of functionality. At the center of the testbench is the Design Under Test (DUT). It is connected to a layer of transactors (drivers, monitors, responders). The transactors communicate with the DUT at the pin level by driving and sampling DUT signals, and with the rest of the UVM testbench by passing transaction objects. They convert data between pins and transactions, i.e. from/to signal to/from transaction level. The testbench layer above the transactor layer consists of components that interact exclusively at the transaction level, such as scoreboards, coverage collectors, stimulus generators, etc.

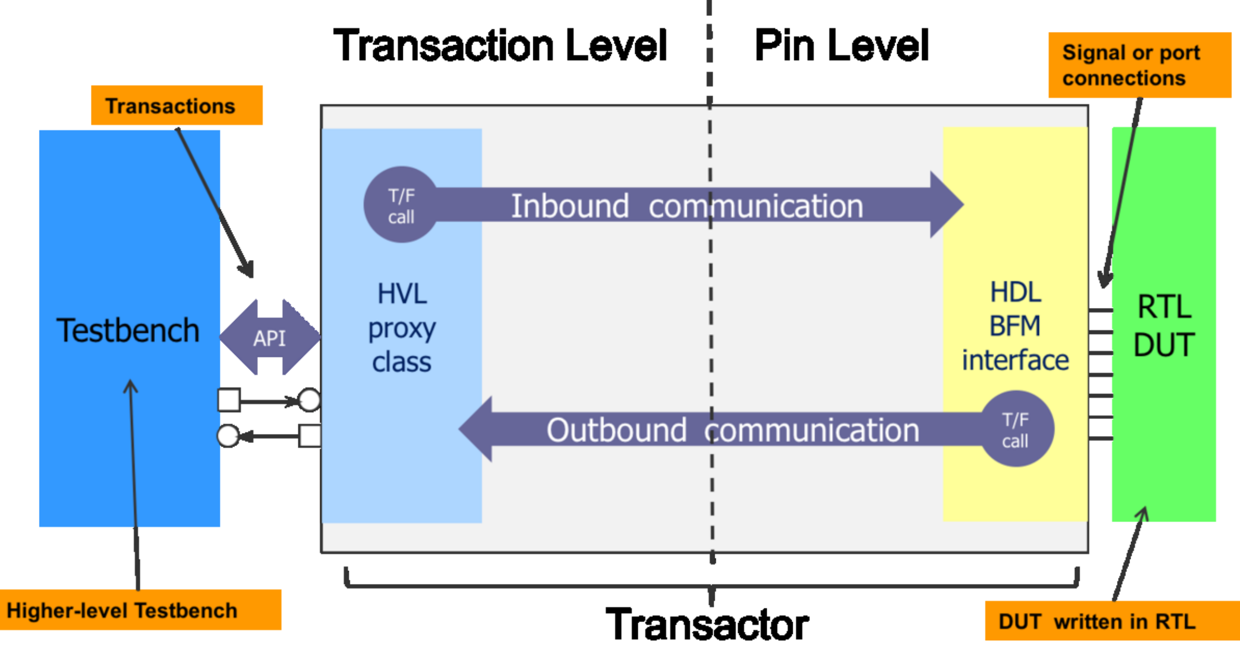


The transactor and testbench layers are conventionally built purely from SystemVerilog classes. However, that construction style limits portability by only targeting a SystemVerilog simulator. A somewhat alternate style, or architecture, that utilizes SystemVerilog classes as well as SystemVerilog interfaces, can increase portability between execution engines. Taking advantage of both these SystemVerilog constructs, a natural split can be made in the transactor layer between transaction level communication grouped on one side separate from timed, signal level communication grouped on the other side.



Partitioned Transactors [1]

The separation of different abstraction levels grouped into like functionality leads to a Dual Top partitioned testbench architecture where one top level contains all of the timed, signal level code in a so-called HDL domain, while the other so-called HVL/TB top contains all of the transaction level testbench domain code. Since transactors typically lie in both domains, they must also be paritioned into two parts. These two parts are a BFM interface and a proxy class. The BFM interface handles signal level code while the proxy class handles anything else that a conventional transactor would do. The BFM and proxy communicate with each other through function and task calls.



While this dual top testbench architecture enables portability [1], it also reduces modeling flexibility to a degree primarily because the signal level code is placed into a SystemVerilog interface instead of a class. SystemVerilog classes offer powerful object-oriented capabilities including inheritance and polymorphism that SystemVerilog interfaces forego.

The UVM Testbench Build and Connection Process

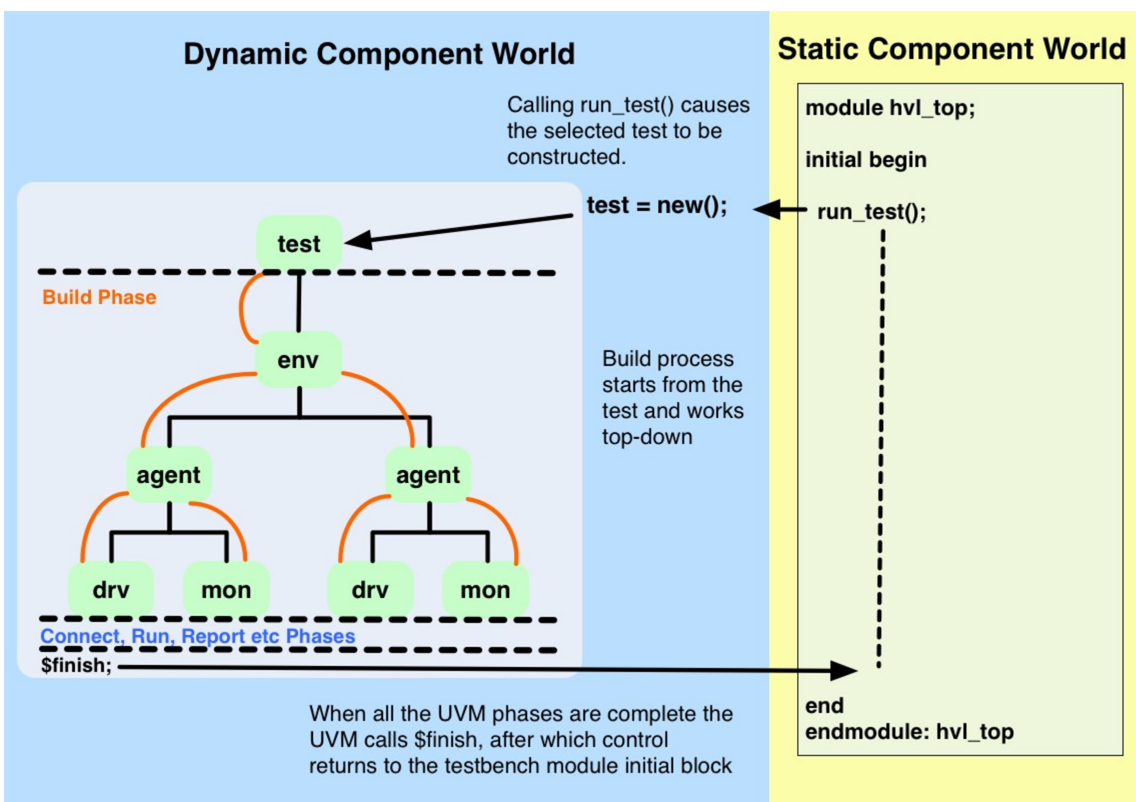
The process to configure and build all the layers of a dual top portable testbench is described in the article on building UVM testbenches. Examples are provided to show how to build a block-level testbench and how to integrate multiple block-level testbenches into higher-level testbench.

References

[1] https://verificationacademy.com/patterns-library/implementation-patterns/environment-patterns/dual-domain-hierarchy-pattern

Building a UVM Testbench

The first phase of a 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 constructed and configured before the next level down (sometimes also referred to as deferred construction).



The UVM Build flow in the context of the testbench

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

During the execution of the test's build phase, the various testbench component configuration objects are prepared and the virtual interfaces in these configuration objects are assigned to the associated testbench interfaces. The configuration objects are then put into the UVM configuration database for this test. Subsequently, the next level of hierarchy is built.

At the next level of hierarchy the corresponding configuration object prepared at the previous level is retrieved and further configuration may take place. Before this configuration object is used to guide the construction and configuration of the next level of hierarchy, it may be modified at the current level of hierarchy. Such conditional

construction thus 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 leaf level is reached.

After the build phase completes, the connect phase commences to make all inter-component connections. Contrary to the build phase, the connect phase works bottom-up from the leafs to the top of the testbench hierarchy. Following the connect phase, the rest of the UVM phases run to completion (also bottom-up) upon which control is passed back to the testbench module.

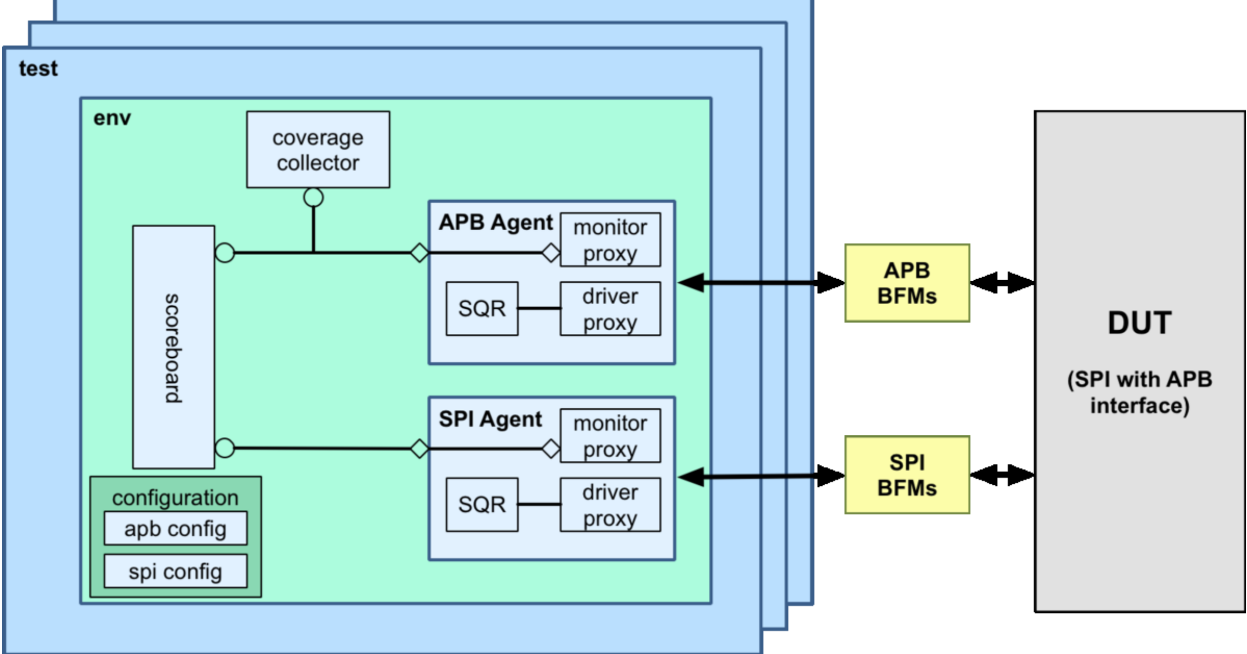
The Test is The Starting Point for The Build Process

The build process for a UVM testbench starts from the test class and works top-down. The test class build method is the first one called at the build phase and it (i.e. the method implementation) determines what gets built in a UVM testbench. Its function is to:

- Set up any factory overrides so that configuration objects or component objects are created as derived types as needed
- Create and configure the configuration objects required by the various sub-components
- Assign the virtual interface handles put into configuration space by the HDL testbench module
- Build up an encapsulating env configuration object and include it into the configuration space
- Build the next level down, usually the top-level env, in the testbench hierarchy

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

Shown below is a block level verification environment to help explain concretely how the test build process works. This is an environment for a SPI master interface DUT, which contains two agents, one for its APB bus interface and one for its SPI interface. A detailed account of the build and connect phases for this example is found in the Block Level Testbench Example article.



Factory Overrides

The UVM factory allows a UVM class to be substituted with another derived class at the point of construction. This can be useful for specializing (i.e. customizing or extending) component behavior or 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 grouping component like an agent or env should have a configuration object that defines its structure and behavior. These configuration objects should be created in the build method of the test and implemented to fit the requirements of the test case. If the configuration of a sub-component is either complex or likely to change, it is advised to add a virtual function implementing the basic (or default) configuration handling, which can then be overwritten in test cases extending from the base test class by overloading the virtual function.

```
//
// 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_apb_agent(apb_agent_config cfg);
```

```
extern function void set_seqs(spi_vseq_base seq);

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
function void spi_test_base::build_phase(uvm_phase phase);
  // env configuration
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // APB configuration
  m_apb_cfg = apb_agent_config::type_id::create("m_apb_cfg");
  configure_apb_agent(m_apb_cfg);
  m_env_cfg.m_apb_agent_cfg = m_apb_cfg;
  // The SPI is not configured as such
  m_spi_cfg.has_functional_coverage = 0;
  m_env_cfg.m_spi_agent_cfg = m_spi_cfg;
  uvm_config_db #(spi_env_config)::set(this, "*", "spi_env_config",
m_env_cfg);
  m_env = spi_env::type_id::create("m_env", this);
endfunction: build_phase

function void spi_test_base::set_seqs(spi_vseq_base seq);
  seq.m_cfg = m_env_cfg;
  seq.spi = m_env.m_spi_agent.m_sequencer;
endfunction
```

Assigning Virtual Interfaces From The Configuration Space

Before calling the UVM *run_test()* method, the links to the signals at the top level I/O boundary of the DUT must be made by connecting them to SystemVerilog interfaces to create a pin interface. The pin interface is then connected to the (driver and monitor) BFM interfaces and a handle to each BFM interface is assigned to a virtual interface handle which is passed to the test through a uvm_config_db::set call. See the article on virtual interfaces for more detailed information.

In the *build()* method of the test these virtual interface handles are to be assigned to the virtual interface handles inside the pertinent component configuration object(s). Individual components then access the virtual interface handle inside their configuration object to drive or monitor the DUT via method calls. In order to keep components modular and reusable, drivers and monitors should not retrieve their virtual interface handles directly from the configuration space, but 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 the use of the uvm_config_db::get method in the SPI testbench example to make virtual interface assignments to the virtual interface handles in the apb_agent configuration object:

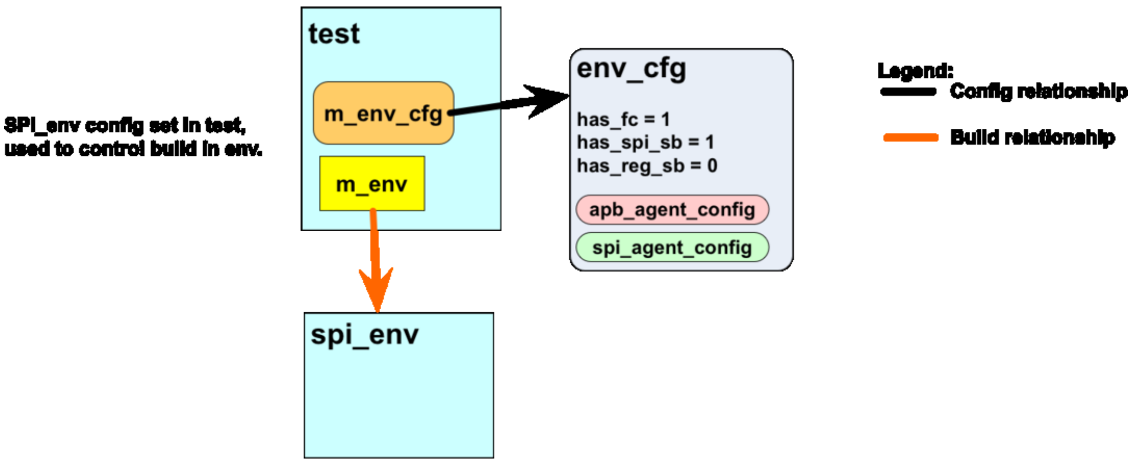
```
// The build method from earlier, adding the apb agent virtual
interface assignment
// Build the env, create the env configuration including any sub
configurations and assign virtual interfaces
function void spi_test_base::build_phase( uvm_phase phase );
  // Create env configuration object
  m_env_cfg = 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);
  // Add the APB driver BFM virtual interface
  if ( !uvm_config_db #(virtual apb_driver_bfm)::get(this, "", "APB_drv_bfm",
m_apb_cfg.drv_bfm ) ) `uvm_error(...)
  // Add the APB monitor BFM virtual interface
  if ( !uvm_config_db #(virtual apb_monitor_bfm)::get(this, "", "APB_mon_bfm",
m_apb_cfg.mon_bfm ) ) `uvm_error(...)
    ...
endfunction: build_phase
```

Embedding 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 embed the configuration objects inside one another in a way that reflects the hierarchy itself. At each intermediate level in the testbench the configuration object for that level is "unfolded" to yield its sub-configuration objects which are re-configured (if necessary) and then passed to the relevant sub-components using uvm_config_db::set.

Following the SPI block level environment example, each of the agents has a separate configuration object. The env configuration object has a handle to each agent configuration object. In the test, all three configuration objects are constructed and configured from a test case viewpoint, and the agent configuration objects are assigned to the corresponding agent configuration object handles inside the env configuration object. Subsequently the env configuration object is added to the configuration space to be retrieved later when the env is built.



Testbench Build Process – Step 1 – At the end of the test build() method

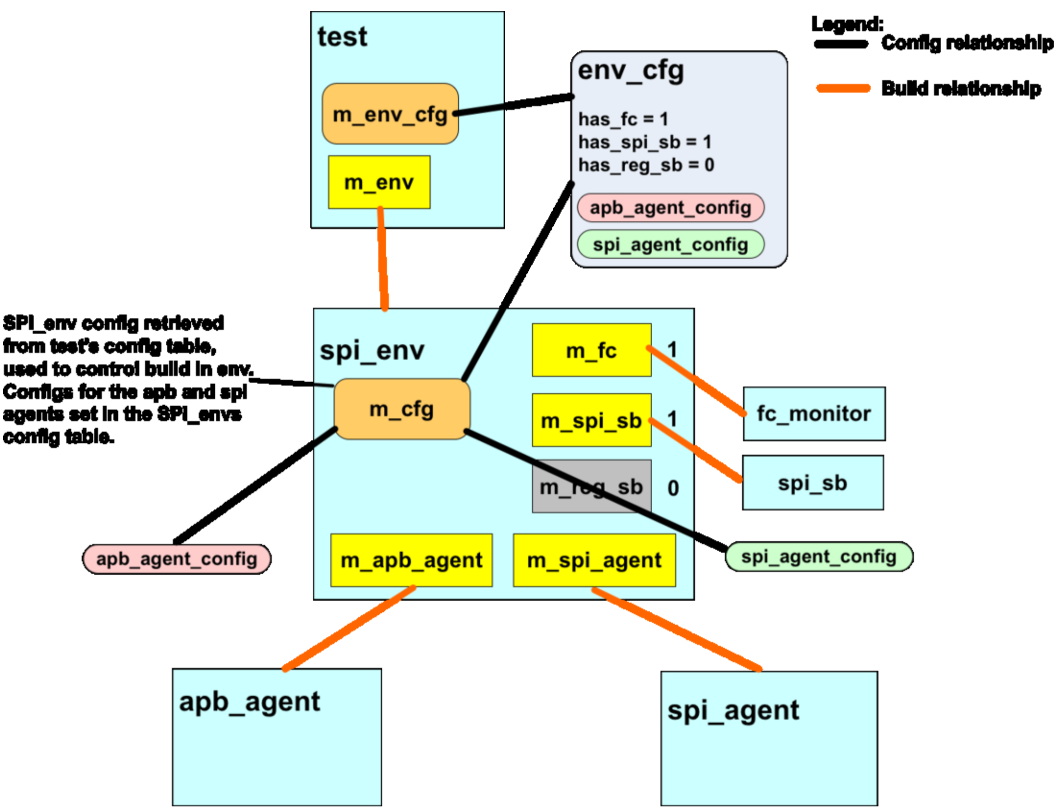
For more complex environments additional levels of encapsulation are required.

```
//
// 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 = 0;
  bit has_spi_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
//------------------------------------------

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 earlier, adding the apb agent virtual
interface assignment
// Build the env, create the env configuration including any sub
configurations and assign 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 monitor BFM virtual interface:
if ( !uvm_config_db #(virtual apb_monitor_bfm)::get(this, "", "APB_mon_bfm",
m_apb_cfg.mon_bfm ) ) `uvm_error(...)
  // Adding the APB driver BFM virtual interface:
if ( !uvm_config_db #(virtual apb_driver_bfm)::get(this, "", "APB_drv_bfm",
m_apb_cfg.drv_bfm ) ) `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);
  // Adding the SPI driver BFM virtual interface
  if ( !uvm_config_db #(virtual spi_driver_bfm)::get(this, "", "SPI_drv_bfm",
m_spi_cfg.drv_bfm ) ) `uvm_error(...)
  // Adding the SPI monitor BFM virtual interface
  if ( !uvm_config_db #(virtual spi_monitor_bfm)::get(this, "", "SPI_mon_bfm",
m_spi_cfg.mon_bfm ) ) `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 , "*", "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 between several envs.



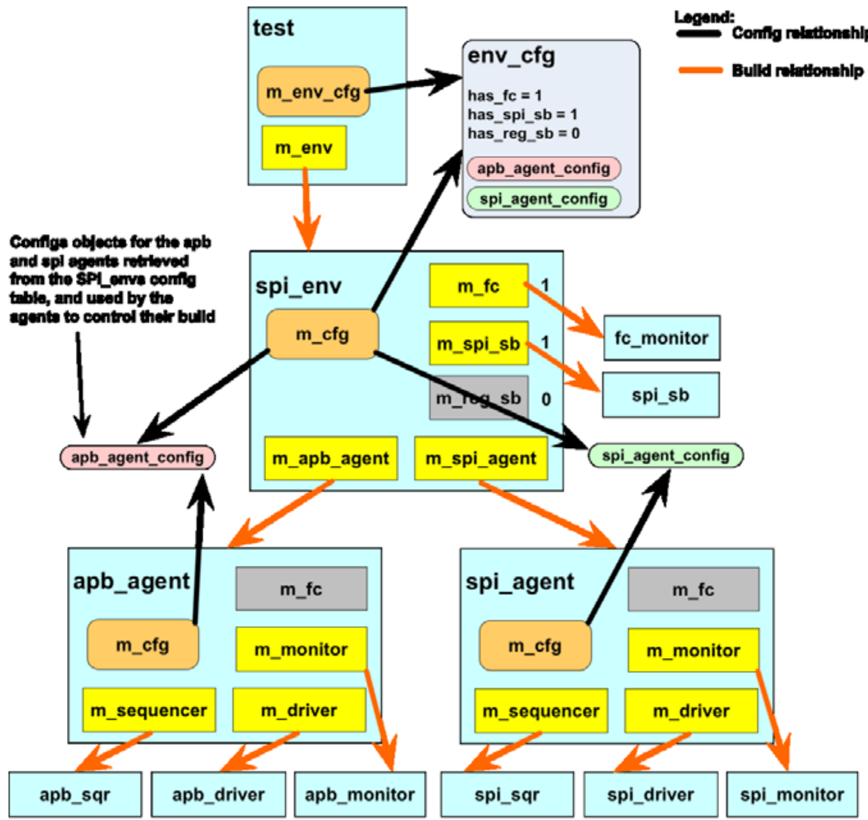
Testbench Build Process – Step 2 – At the end of spi_envs build() method

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 references the uvm_component in which it is created. Using the same name as the handle facilitates cross-referencing paths and handles. For instance, in the previous code snippet, the spi_env is created in the test using its declaration handle m_env, and hence following the build process the UVM "dynamic path" name to this spi_env is "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 followed by the *build()* method of each of its children, and so forth, until the full environment hierarchy has been constructed. This deferred approach to construction enables each *build()* method to affect what happens in the build process of components at lower levels in the hierarchy. For instance, if an agent is configured to be passive, the build process for the agent omits the creation of the agent's sequencer and driver since these are only required if the agent is active.



Testbench Build Process – Stage 3 - End of Agent build() method

The Hierarchical Connection Process

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

Configuration objects are once again at play during the connection process as they may contain references to virtual interfaces or other 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 are only made if the agent is active.

Examples

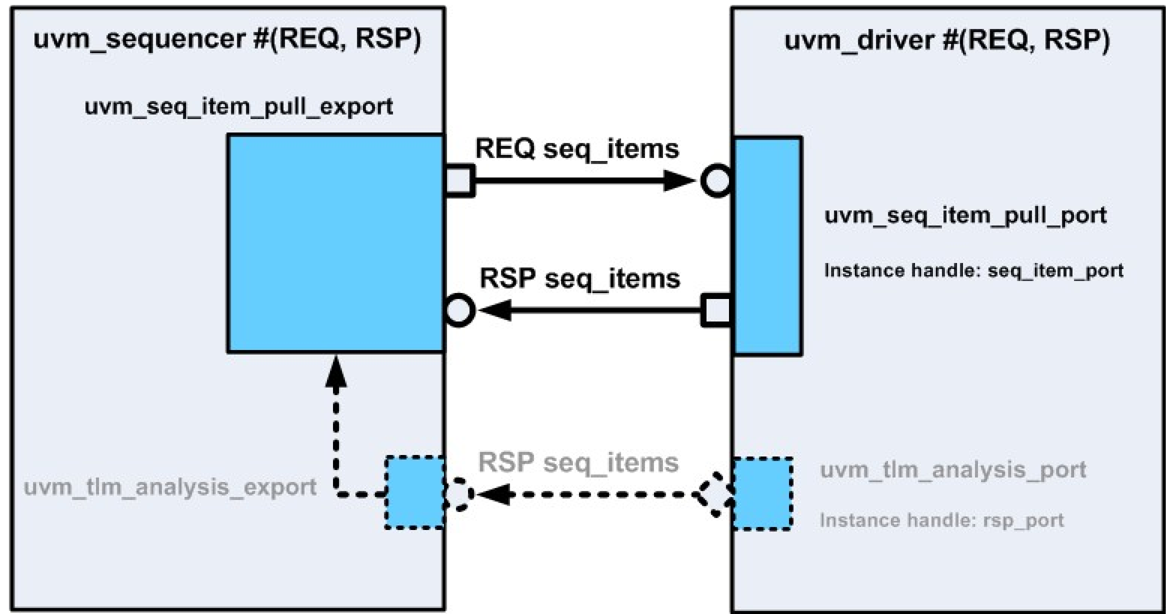
The UVM build process is best illustrated through some examples that illustrate how different component hierarchies are built up:

A block level testbench containing an agent

An integration level testbench

Sequencer-Driver Connections|Connecting the Sequencer and Driver

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 parameterized 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.



Sequencer-Driver Connections

The connection between the driver port and the sequencer export is made using a TLM connect method during the connect phase:

```
// Driver parameterized with the same sequence_item for request &
response
// response defaults to request
class adpcm_driver extends uvm_driver #(adpcm_seq_item);
....
endclass: adpcm_driver

// Agent containing a driver and a sequencer - uninteresting bits left
out
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
adpcm_agent_config m_cfg;
// uvm_sequencer parameterized with the adpcm_seq_item for request &
response
uvm_sequencer #(adpcm_seq_item) m_sequencer;
```

```
// 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
```

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:

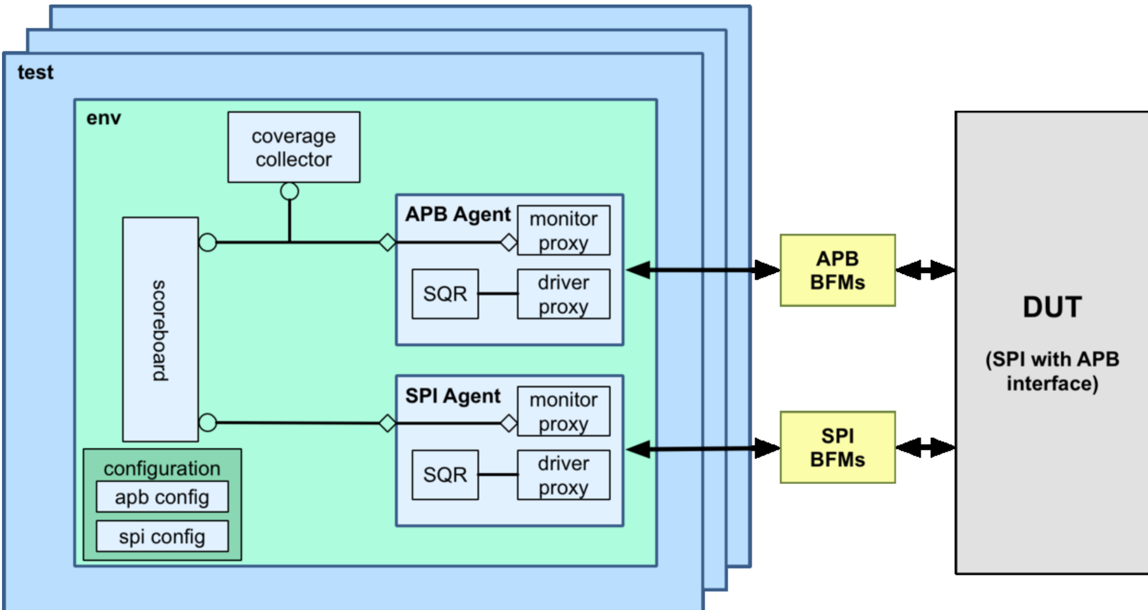
```
// Same agent as in the previous bidirectional example:
class adpcm_agent extends uvm_agent;

adpcm_driver m_driver;
uvm_sequencer #(adpcm_seq_item) 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.

Block-Level Testbench

As an example of a block level testbench, consider a testbench 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. Let us go through each layer of the testbench and describe how it is put together from the top down.



The Testbench Modules

Two top level testbench modules are used in the SPI block level testbench. The hdl_top module contains the SPI Master DUT, the APB and SPI BFMs and the apb_if, spi_if and intr_if pin interfaces. The SPI Master DUT is connected to the apb_if, spi_if and intr_if which in turn are connected to the APB and SPI master and slave BFMs, respectively. Two initial blocks are also encapsulated within the hdl_top module. The first initial block places the virtual interface handles for the BFM interfaces into the UVM configuration space using uvm_config_db::set. The second initial block generates a clock and a reset signal for the APB interface.

```
module hdl_top;

`include "timescale.v"

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

//
// Instantiate the pin interfaces:
//
apb_if APB(PCLK, PRESETn);
spi_if SPI();
intr_if INTR();
```

```
//
// Instantiate the BFM interfaces:
//
    apb_monitor_bfm APB_mon_bfm(
     .PCLK       (APB.PCLK),
     .PRESETn (APB.PRESETn),
     .PADDR      (APB.PADDR),
     .PRDATA     (APB.PRDATA),
     .PWDATA     (APB.PWDATA),
     .PSEL       (APB.PSEL),
     .PENABLE (APB.PENABLE),
     .PWRITE     (APB.PWRITE),
     .PREADY     (APB.PREADY)
);
    apb_driver_bfm APB_drv_bfm(
     .PCLK       (APB.PCLK),
     .PRESETn (APB.PRESETn),
     .PADDR      (APB.PADDR),
     .PRDATA     (APB.PRDATA),
     .PWDATA     (APB.PWDATA),
     .PSEL       (APB.PSEL),
     .PENABLE (APB.PENABLE),
     .PWRITE     (APB.PWRITE),
     .PREADY     (APB.PREADY)
);
spi_monitor_bfm SPI_mon_bfm(
     .clk      (SPI.clk),
     .cs       (SPI.cs),
     .miso (SPI.miso),
     .mosi (SPI.mosi)
);
spi_driver_bfm SPI_drv_bfm(
     .clk      (SPI.clk),
     .cs       (SPI.cs),
     .miso (SPI.miso),
     .mosi (SPI.mosi)
);
intr_bfm INTR_bfm(
     .IRQ      (INTR.IRQ),
     .IREQ (INTR.IREQ)
);

// 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)
);


// Initial block for virtual interface wrapping:
initial begin
    import uvm_pkg::uvm_config_db;
    uvm_config_db #(virtual apb_monitor_bfm)::set(null, "uvm_test_top", "APB_mon_bfm",
APB_mon_bfm);
    uvm_config_db #(virtual apb_driver_bfm) ::set(null, "uvm_test_top", "APB_drv_bfm",
APB_drv_bfm);
    uvm_config_db #(virtual spi_monitor_bfm)::set(null, "uvm_test_top", "SPI_mon_bfm",
SPI_mon_bfm);
    uvm_config_db #(virtual spi_driver_bfm) ::set(null, "uvm_test_top", "SPI_drv_bfm",
SPI_drv_bfm);
    uvm_config_db #(virtual intr_bfm)                    ::set(null, "uvm_test_top",
"INTR_bfm", INTR_bfm);
end

//
// Initial blocks for clock and reset generation:
//
initial begin
    PCLK = 0;
    forever #10ns PCLK = ~PCLK;
end
initial begin
    PRESETn = 0;
    repeat(4) @(posedge PCLK);
    PRESETn = 1;
end
```

```
endmodule: hdl_top
```

The hvl_top module simply imports the uvm_pkg and the spi_test_lib_pkg which contains definitions for the tests that can be run. It also contains the initial block that calls the run_test() method to construct and launch the specified test and thus UVM phasing.

```
module hvl_top;

`include "timescale.v"

import uvm_pkg::*;
import spi_test_lib_pkg::*;

// UVM initial block:
initial begin
  run_test();
end

endmodule: hvl_top
```

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, after first creating and preparing all pertinent configuration objects to be used by the environment. The configuration and build process is largely common to most test cases, so it is generally good practice to devise 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 required sub-configuration objects to the env, as part of the build method of the spi_env. This "Russian Doll" approach to nesting configurations 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 themselves constructed, and their virtual interfaces assigned using the uvm_config_db::get method, and then configured. The virtual interface assignments are to the virtual BFM interface handles that were set in the hdl_top. The APB agent may well be configured differently for different test cases and so its configuration process has been dedicated to a specific virtual method in the base class. This lets derived test classes overload this method and custom configure the APB agent as required.

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;
//Interrupt Utility
intr_util INTR;

//-----------------------------------------
// 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 virtural interfaces
function void spi_test_base::build_phase( uvm_phase phase );
  virtual intr_bfm temp_intr_bfm;
  // env configuration
  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_monitor_bfm)::get(this, "", "APB_mon_bfm",
m_apb_cfg.mon_bfm)) `uvm_fatal(...)
```

```
  if (!uvm_config_db #(virtual apb_driver_bfm) ::get(this, "", "APB_drv_bfm",
m_apb_cfg.drv_bfm)) `uvm_fatal(...)
  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_monitor_bfm)::get(this, "", "SPI_mon_bfm",
m_spi_cfg.mon_bfm)) `uvm_fatal(...)
  if (!uvm_config_db #(virtual spi_driver_bfm) ::get(this, "", "SPI_drv_bfm",
m_spi_cfg.drv_bfm)) `uvm_fatal(...)
  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:
  INTR = intr_util::type_id::create("INTR");
  if (!uvm_config_db #(virtual intr_bfm)::get(this, "", "INTR_bfm", temp_intr_bfm) )
                                   `uvm_fatal(...)
  INTR.set_bfm(temp_intr_bfm);
  m_env_cfg.INTR = INTR;

  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. As a result, the test writer only needs to add a run_phase method. In the following (simplistic and to be updated) example, the run_phase method instantiates sequences and starts them on appropriate sequencers in the env. All of the configuration process is carried out by the super.build_phase() method call in the build_phase method.

```
//
// Class Description:
//
//
class spi_poll_test extends spi_test_base;

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

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

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

endclass: spi_poll_test

function spi_poll_test::new(string name = "spi_poll_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_poll_test::build_phase(uvm_phase phase);
  super.build_phase(phase);
endfunction: build_phase

task spi_poll_test::run_phase(uvm_phase phase);

  config_polling_test t_seq =
config_polling_test::type_id::create("t_seq");
  set_seqs(t_seq);

  phase.raise_objection(this, "Test Started"); t_seq.start(null);
  #100;
  phase.drop_objection(this, "Test Finished");

endtask: run_phase
```

The Environment

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 and a functional coverage collector. 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 utility which contains a method for detecting an interrupt. This will be used by ***sequences***. The contents of the spi_env_config class are as follows:

```
//
// Class Description:
//
//
class spi_env_config extends uvm_object;

const string s_my_config_id = "spi_env_config";
const string s_no_config_id = "no config";
const string s_my_config_type_error_id = "config type error";

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

// Interrupt Utility - used in the wait for interrupt task
//
intr_util 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;
 INTR.wait_for_interrupt();
endtask: wait_for_interrupt

// Check that interrupt has cleared:
function bit spi_env_config::is_interrupt_cleared;
 return INTR.is_interrupt_cleared();
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_scoreboard m_scoreboard;

  // Register layer adapter
```

```
  reg2apb_adapter m_reg2apb;
  // Register predictor
  uvm_reg_predictor#(apb_seq_item) m_apb2reg_predictor;

  //------------------------------------------
  // 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))
    `uvm_fatal("CONFIG_LOAD", "Cannot get() configuration spi_env_config
from uvm_config_db. Have you set() it?")

  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);

  // Build the register model predictor
  m_apb2reg_predictor = uvm_reg_predictor#(apb_seq_item)::type_id::create("m_apb2reg_predictor",
 this);
  m_reg2apb = reg2apb_adapter::type_id::create("m_reg2apb");

  uvm_config_db #(spi_agent_config)::set(this, "m_spi_agent*",
                                         "spi_agent_config",
                                         m_cfg.m_spi_agent_cfg);
  m_spi_agent = spi_agent::type_id::create("m_spi_agent", this);
```

```
    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);

    // Only set up register sequencer layering if the spi_rb is the top
block
    // If it isn't, then the top level environment will set up the
correct sequencer
    // and predictor
    if(m_cfg.spi_rb.get_parent() == null) begin if(m_cfg.m_apb_agent_cfg.active
        == UVM_ACTIVE) begin

m_cfg.spi_rb.spi_reg_block_map.set_sequencer(m_apb_agent.m_sequencer, m_reg2apb);
        end

        //
        // 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:
        m_apb2reg_predictor.map = m_cfg.spi_rb.spi_reg_block_map;
        // Set the predictor adapter:
        m_apb2reg_predictor.adapter = m_reg2apb;
        // Disable the register models auto-prediction
        m_cfg.spi_rb.spi_reg_block_map.set_auto_predict(0);
        // Connect the predictor to the bus agent monitor analysis port
        m_apb_agent.ap.connect(m_apb2reg_predictor.bus_in);
    end

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

endfunction: connect_phase
```

The Agents

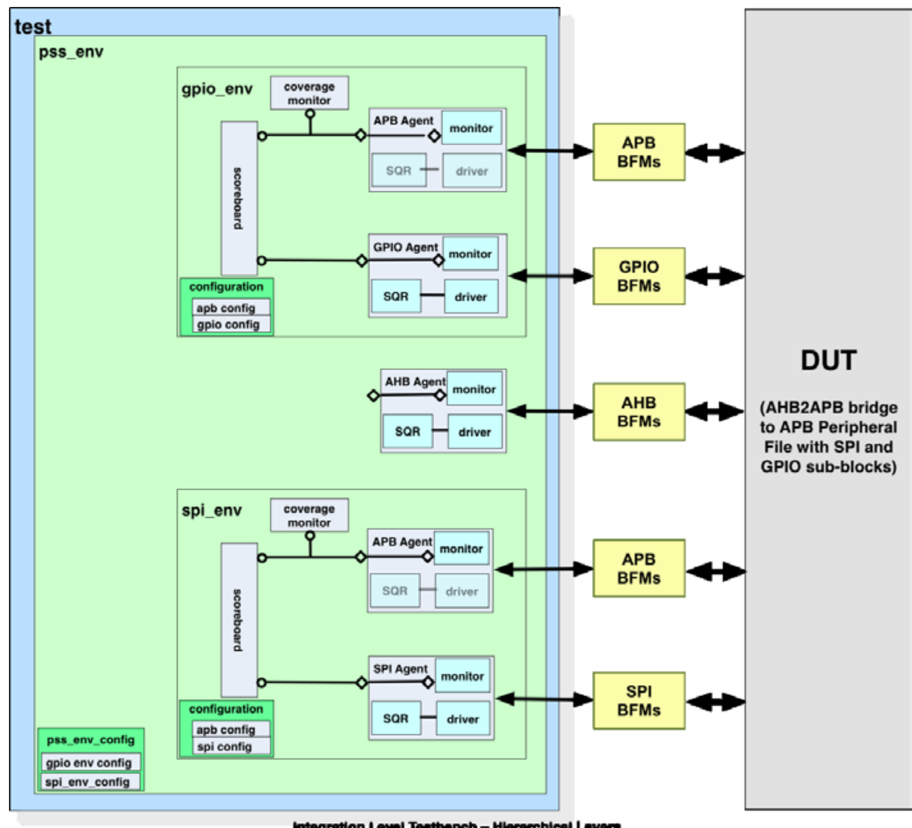
Since the UVM build process is top down, the SPI and APB agents are constructed next. The article on the ***agent build process*** describes how the APB agent is configured and built, and the SPI agent follows the same process.

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

Integration-Level Testbench

This testbench 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.



Integration Level Testbench – Hierarchical Layers

We shall now go through the testbench and the build process from the top down, starting with the two top level testbench modules.

Top Level Testbench Modules

As with the block level testbench example, two top level modules are utilized. The hdl_top instantiates the DUT, instantiates the BFM interfaces and connects the pin interfaces to the DUT and the BFM interfaces. Virtual Interface handles referencing the BFM interfaces are placed into the configuration space and clocks and resets are generated. The main differences between this code and the block level testbench code are that there are more interfaces and that there is a need to bind to some internal signals to monitor the APB bus. Another differences is that driver BFMs are not instantiated if the agent is going to be used in passive mode. The DUT is wrapped by a module which connects its I/O signals to the interfaces used in the UVM testbench. 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 pin interfaces:
//
apb_if APB(HCLK, HRESETn); // APB interface - shared between passive
agents
ahb_if AHB(HCLK, HRESETn);   // AHB interface
spi_if SPI();  // SPI Interface
...  //Additional pin interfaces

//
// Instantiate the BFM interfaces:
//
apb_monitor_bfm APB_SPI_mon_bfm(
   .PCLK    (APB.PCLK),
   .PRESETn (APB.PRESETn),
   .PADDR   (APB.PADDR),
   .PRDATA  (APB.PRDATA),
   .PWDATA  (APB.PWDATA),
   .PSEL    (APB.PSEL),
   .PENABLE (APB.PENABLE),
   .PWRITE  (APB.PWRITE),
   .PREADY  (APB.PREADY)
);
apb_monitor_bfm APB_GPIO_mon_bfm(
   .PCLK    (APB.PCLK),
   .PRESETn (APB.PRESETn),
   .PADDR   (APB.PADDR),
   .PRDATA  (APB.PRDATA),
```

```
        .PWDATA   (APB.PWDATA),
        .PSEL     (APB.PSEL),
        .PENABLE (APB.PENABLE),
        .PWRITE   (APB.PWRITE),
        .PREADY   (APB.PREADY)
);
        apb_driver_bfm APB_GPIO_drv_bfm(
        .PCLK     (APB_dummy.PCLK),
        .PRESETn (APB_dummy.PRESETn),
        .PADDR    (APB_dummy.PADDR),
        .PRDATA   (APB_dummy.PRDATA),
        .PWDATA   (APB_dummy.PWDATA),
        .PSEL     (APB_dummy.PSEL),
        .PENABLE (APB_dummy.PENABLE),
        .PWRITE   (APB_dummy.PWRITE),
        .PREADY   (APB_dummy.PREADY)
);
...     //Additional BFM interfaces

// 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
initial begin
   import uvm_pkg::uvm_config_db;
   uvm_config_db #(virtual apb_monitor_bfm)            ::set(null, "uvm_test_top",
 "APB_SPI_mon_bfm", APB_SPI_mon_bfm);
   uvm_config_db #(virtual apb_monitor_bfm)            ::set(null, "uvm_test_top",
 "APB_GPIO_mon_bfm", APB_GPIO_mon_bfm);
   uvm_config_db #(virtual ahb_monitor_bfm)            ::set(null, "uvm_test_top",
 "AHB_mon_bfm", AHB_mon_bfm);
   uvm_config_db #(virtual ahb_driver_bfm)             ::set(null, "uvm_test_top",
 "AHB_drv_bfm", AHB_drv_bfm);
   uvm_config_db #(virtual spi_monitor_bfm)            ::set(null, "uvm_test_top",
```

```
 "SPI_mon_bfm", SPI_mon_bfm);
  uvm_config_db #(virtual spi_driver_bfm)   ::set(null, "uvm_test_top",
 "SPI_drv_bfm", SPI_drv_bfm);
  ...  //Additional uvm_config_db::set() calls
end

//
// Clock and reset initial block:
//
initial begin
  HCLK = 1;
  forever #10ns HCLK = ~HCLK;
end
initial begin
  HRESETn = 0;
  repeat(4) @(posedge HCLK);
  HRESETn = 1;
end

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

endmodule: hdl_top
```

The hvl_top module remains largely the same as in the block level example. It now imports the pss_test_lib_pkg so that the definition of the tests are known. Otherwise, the same functionality is present.

```
module hvl_top;

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

// UVM initial block:
initial begin
  run_test();
end

endmodule: hvl_top
```

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 un-nesting 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;
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;
... //Additional configuration object handles

// Register map
pss_register_map pss_rm;

//Interrupt Utility
intr_util ICPIT;

//------------------------------------------
```

```
// 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 virtural interfaces
function void pss_test_base::build_phase(uvm_phase phase);
   virtual intr_bfm temp_intr_bfm;

   m_env_cfg = pss_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);

   // Register map - Keep reg_map a generic name for vertical reuse
reasons
   pss_rb = pss_reg_block::type_id::create("pss_rb"); pss_rb.build();
   m_env_cfg.pss_rb = pss_rb;

   // SPI Sub-env configuration:
   m_spi_env_cfg = spi_env_config::type_id::create("m_spi_env_cfg"); m_spi_env_cfg.spi_rb =
   pss_rb.spi_rb;

   // apb agent in the SPI env:

   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_monitor_bfm)::get(this, "", "APB_SPI_mon_bfm",
m_spi_apb_agent_cfg.mon_bfm))
                    `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface
```

```
APB_SPI_mon_bfm from uvm_config_db. Have you set() it?")
    //if (!uvm_config_db #(virtual apb_driver_bfm) ::get(this, "",
"APB_SPI_drv_bfm", m_spi_apb_agent_cfg.drv_bfm))
    //      `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface
APB_SPI_drv_bfm from uvm_config_db. Have you set() it?")
    m_spi_apb_agent_cfg.active = UVM_PASSIVE;
    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_monitor_bfm)::get(this, "", "SPI_mon_bfm",
m_spi_agent_cfg.mon_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_mon_bfm from uvm_config_db.
Have you set() it?")
    if (!uvm_config_db #(virtual spi_driver_bfm) ::get(this, "", "SPI_drv_bfm",
m_spi_agent_cfg.drv_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm from uvm_config_db.
Have you set() it?")
    m_spi_env_cfg.m_spi_agent_cfg = m_spi_agent_cfg;
    m_env_cfg.m_spi_env_cfg = m_spi_env_cfg;
    uvm_config_db #(spi_env_config)::set(this, "*", "spi_env_config", m_spi_env_cfg);

    // GPIO env configuration:
    m_gpio_env_cfg = gpio_env_config::type_id::create("m_gpio_env_cfg"); m_gpio_env_cfg.gpio_rb =
    pss_rb.gpio_rb;
    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_monitor_bfm)::get(this, "", "APB_GPIO_mon_bfm",
m_gpio_apb_agent_cfg.mon_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface
APB_GPIO_mon_bfm from uvm_config_db. Have you set() it?")
    //if (!uvm_config_db #(virtual apb_driver_bfm) ::get(this, "",
"APB_GPIO_drv_bfm", m_gpio_apb_agent_cfg.drv_bfm))
    //      `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface APB_drv_bfm
from uvm_config_db. Have you set() it?") m_gpio_apb_agent_cfg.active =
    UVM_PASSIVE; 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");
    if (!uvm_config_db #(virtual gpio_monitor_bfm)::get(this, "",
```

```
"GPO_mon_bfm", m_GPO_agent_cfg.mon_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface GPO_mon_bfm from
uvm_config_db. Have you set() it?")
    //if (!uvm_config_db #(virtual gpio_driver_bfm) ::get(this, "",
"GPO_drv_bfm", m_GPO_agent_cfg.drv_bfm))
    //      `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm
from uvm_config_db. Have you set() it?") m_GPO_agent_cfg.active =
    UVM_PASSIVE; // Only monitors m_gpio_env_cfg.m_GPO_agent_cfg =
    m_GPO_agent_cfg;

    // GPOE agent
    m_GPOE_agent_cfg =
gpio_agent_config::type_id::create("m_GPOE_agent_cfg");
    if (!uvm_config_db #(virtual gpio_monitor_bfm)::get(this, "", "GPOE_mon_bfm",
m_GPOE_agent_cfg.mon_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface GPOE_mon_bfm from
uvm_config_db. Have you set() it?")
    //if (!uvm_config_db #(virtual gpio_driver_bfm) ::get(this, "",
"GPOE_drv_bfm", m_GPOE_agent_cfg.drv_bfm))
    //      `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm
from uvm_config_db. Have you set() it?") m_GPOE_agent_cfg.active =
    UVM_PASSIVE; // Only monitors m_gpio_env_cfg.m_GPOE_agent_cfg =
    m_GPOE_agent_cfg;

    // GPI agent - active (default)
    m_GPI_agent_cfg =
gpio_agent_config::type_id::create("m_GPI_agent_cfg");
    if (!uvm_config_db #(virtual gpio_monitor_bfm)::get(this, "", "GPI_mon_bfm",
m_GPI_agent_cfg.mon_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface GPI_mon_bfm from uvm_config_db.
Have you set() it?")
    if (!uvm_config_db #(virtual gpio_driver_bfm) ::get(this, "", "GPI_drv_bfm",
m_GPI_agent_cfg.drv_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface SPI_drv_bfm from uvm_config_db.
Have you set() it?")
    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_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_monitor_bfm)::get(this, "", "AHB_mon_bfm",
m_ahb_agent_cfg.mon_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface AHB_mon_bfm from
uvm_config_db. Have you set() it?")
    if (!uvm_config_db #(virtual ahb_driver_bfm) ::get(this, "", "AHB_drv_bfm",
m_ahb_agent_cfg.drv_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() BFM interface AHB_drv_bfm from uvm_config_db.
Have you set() it?")
    m_env_cfg.m_ahb_agent_cfg = m_ahb_agent_cfg;
    // Add in interrupt line
    ICPIT = intr_util::type_id::create("ICPIT");
    if (!uvm_config_db #(virtual intr_bfm)::get(this, "", "ICPIT_bfm", temp_intr_bfm))
        `uvm_fatal("VIF CONFIG", "Cannot get() interface ICPIT_bfm from uvm_config_db. Have
you set() it?")
    ICPIT.set_bfm(temp_intr_bfm);
    m_env_cfg.ICPIT = ICPIT;
    m_spi_env_cfg.INTR = ICPIT;

    uvm_config_db #(pss_env_config)::set(this, "*", "pss_env_config", m_env_cfg);
    m_env = pss_env::type_id::create("m_env", this); 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_spi_polling_test extends pss_test_base;

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

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

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

endclass: pss_spi_polling_test

function pss_spi_polling_test::new(string name = "pss_spi_polling_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_spi_polling_test::build_phase(uvm_phase phase);
    super.build_phase(phase);
endfunction: build_phase

task pss_spi_polling_test::run_phase(uvm_phase phase); config_polling_test
    t_seq =
config_polling_test::type_id::create("t_seq");

    t_seq.m_cfg = m_spi_env_cfg;
    t_seq.spi = m_env.m_spi_env.m_spi_agent.m_sequencer; phase.raise_objection(this,

    "Starting PSS SPI polling test");

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

    phase.drop_objection(this, "Finishing PSS SPI polling 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;

    // Register layer adapter
    reg2ahb_adapter m_reg2ahb;
    // Register predictor
    uvm_reg_predictor#(ahb_seq_item) m_ahb2reg_predictor;

    //------------------------------------------
    // 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(this, "", "pss_env_config", m_cfg) )
        `uvm_fatal("CONFIG_LOAD", "Cannot get() configuration pss_env_config
from uvm_config_db. Have you set() it?")

    uvm_config_db #(spi_env_config)::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);

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

    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);

    // Build the register model predictor
    m_ahb2reg_predictor = uvm_reg_predictor#(ahb_seq_item)::type_id::create("m_ahb2reg_predictor",
  this);
    m_reg2ahb = reg2ahb_adapter::type_id::create("m_reg2ahb");
endfunction: build_phase

// Only required if you have sub-components which are connected
function void pss_env::connect_phase(uvm_phase phase);
    // Only set up register sequencer layering if the pss_rb is the top
block
    // If it isn't, then the top level environment will set up the
correct sequencer
    // and predictor
    if(m_cfg.pss_rb.get_parent() == null) begin if(m_cfg.m_ahb_agent_cfg.active
        == UVM_ACTIVE) begin
            m_cfg.pss_rb.pss_map.set_sequencer(m_ahb_agent.m_sequencer, m_reg2ahb);
        end

        //
        // 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:
    m_ahb2reg_predictor.map = m_cfg.pss_rb.pss_map;
    // Set the predictor adapter:
    m_ahb2reg_predictor.adapter = m_reg2ahb;
    // Disable the register models auto-prediction
    m_cfg.pss_rb.pss_map.set_auto_predict(0);
    // Connect the predictor to the bus agent monitor analysis port
    m_ahb_agent.ap.connect(m_ahb2reg_predictor.bus_in);
  end
endfunction: connect_phase
```

The rest of the testbench hierarchy

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

Further levels of integration

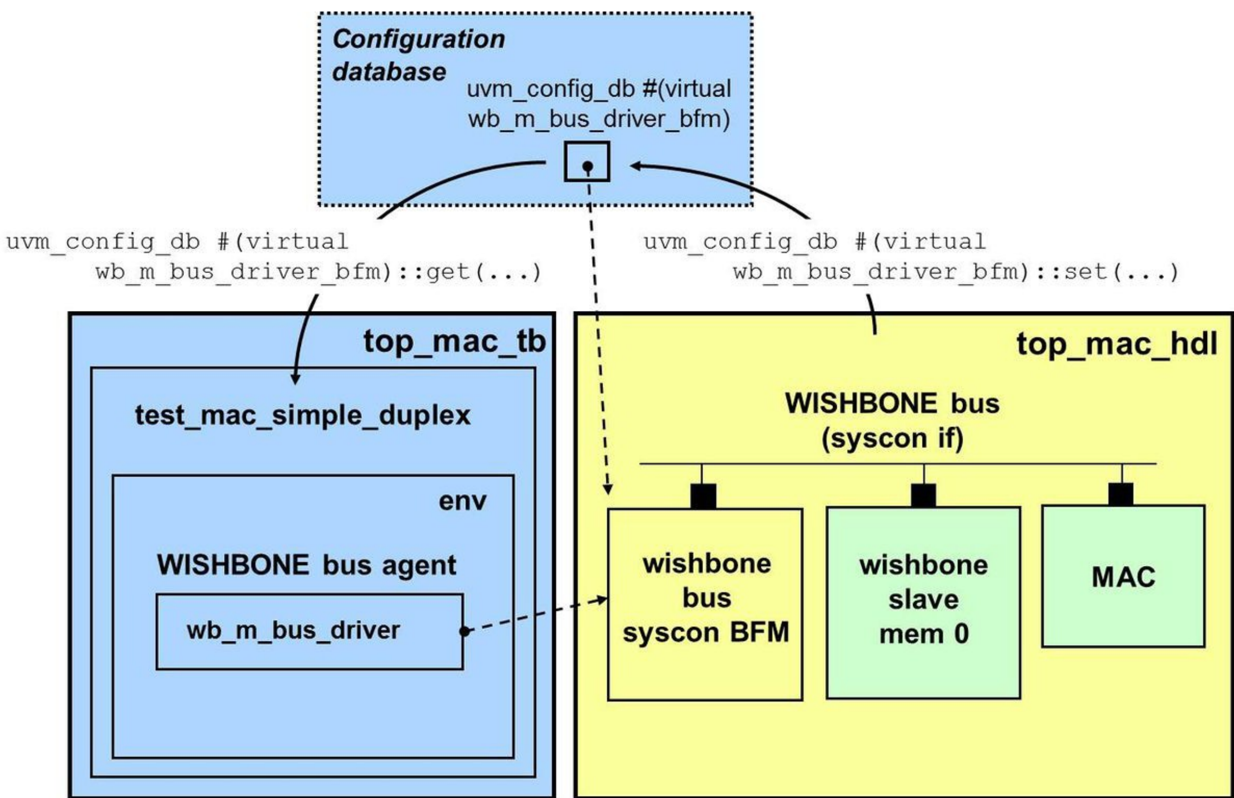
Vertical reuse for further levels of integration can be achieved by extending the process described for the PSS example. Each level of integration adds another layer, so for instance a level 2 integration environment would contain two or more level 1 envs and the level 2 env configuration object would 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.

Dual Top Architecture

The dual top testbench architecture advocated throughout this cookbook enables platform portability - it is fundamental for testbench acceleration using emulation or some other hardware-assisted platform. The HDL top level module encapsulates everything associated directly with the clock cycle-based signal level activity of the RTL DUT, which can be run in simulation or be mapped (i.e. synthesized) onto the emulator. The HVL/TB top level module encapsulates the object-oriented testbench and always runs on the simulator as per usual.

Next to hardware-assisted testbench acceleration there are certainly other good reasons to employ a dual top testbench structure. Specifically, it can facilitate the use of multi-processor platforms for simulation, the use of compile and run-time optimization techniques, or the application of good software engineering practices for the creation of portable, configurable VIP. Having more than one top level module in a simulation complies with the (System)Verilog standard and is quite common indeed. All top level (i.e. uninstantiated) modules are effectively treated as implicit instances at the top of the module hierarchy.

Clearly, the dual HDL and testbench top level module hierarchies must be interconnected, or "bound" together to enable TB - HDL inter-domain communication. This is facilitated using the UVM configuration database as depicted below.



HDL top level module

For the MAC example, the *top_mac_hdl* module contains the MAC DUT along with its associated (signal and BFM) interfaces, a MAC MII wrapper module, and the WISHBONE slave memory with WISHBONE bus logic:

```
module top_mac_hdl;
  import test_params_pkg::*;

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

```
  // BFM interface instances
  wb_m_bus_driver_bfm wb_drv_bfm(wb_bus_if);
  wb_bus_monitor_bfm  wb_mon_bfm(wb_bus_if);

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

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

  initial begin
    // Set BFM interfaces in config space
    import uvm_pkg::uvm_config_db;
    uvm_config_db #(virtual wb_m_bus_driver_bfm)::
      set(null, "uvm_test_top", "WB_DRV_BFM", wb_drv_bfm);
    uvm_config_db #(virtual wb_bus_monitor_bfm)::
      set(null, "uvm_test_top", "WB_MON_BFM", wb_mon_bfm);
  end
endmodule
```

Note the use of the explicitly named import of uvm_config_db local to the initial block to put the BFM interfaces in the UVM config space, which is appropriately discriminate in contrast to the test parameters package wildcard import at the beginning of the *top_mac_hdl* top module. The indiscriminate wild card import on the other hand is appropriate for the test parameters package because multiple elements of that package will generally be used inside *top_mac_hdl*.

Further note that instead of placing the uvm_config_db registration of the BFM interfaces under the HDL top level module, as shown above, this may similarly be placed under the HVL top level module using full cross-domain hierarchical instance paths.

```
initial begin
  // Set BFM interfaces in config space
```

```
  import uvm_pkg::uvm_config_db;
  uvm_config_db #(virtual wb_m_bus_driver_bfm)::
    set(null, "uvm_test_top", "WB_DRV_BFM", top_mac_hdl.wb_drv_bfm);
  uvm_config_db #(virtual wb_bus_monitor_bfm)::
    set(null, "uvm_test_top", "WB_MON_BFM", top_mac_hdl.wb_mon_bfm);
end
```

Putting it on the HDL side is more elegant as the *uvm_config_db::set* calls are localized with the BFM interfaces and relative instance paths can thus be used. Additionally, full instance paths can be computed as strings using the SystemVerilog *%m* string formatter. For instance, one could use *$sformatf("%m.wb_drv_bfm")* instead of *"WB_DRV_BFM"* as the (guaranteed unique) uvm_config_db lookup key for the driver BFM virtual interface.

HVL/TB top level module

The top level testbench module includes an initial block to call the UVM *run_test()* function to launch a test, as shown below. Note that the leading package imports *uvm_pkg::run_test* and *tests_pkg::** are necessary to compile this function call.

```
module top_mac_hvl;
  import uvm_pkg::run_test;
  import tests_pkg::*;

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

  // Optionally the initial block from above for uvm_config_db
  // registration of the HDL side BFM interfaces

endmodule
```

Simulating a Dual Top Testbench

The command line for invoking Questa vsim simply specifies all required top level modules by their module names:

```
#Makefile
...
normal: clean cmp
      vsim -c top_mac_hvl top_mac_hdl
+UVM_TESTNAME=test_mac_simple_duplex -do "run -all; exit"
...
```

DUT-Testbench Connections

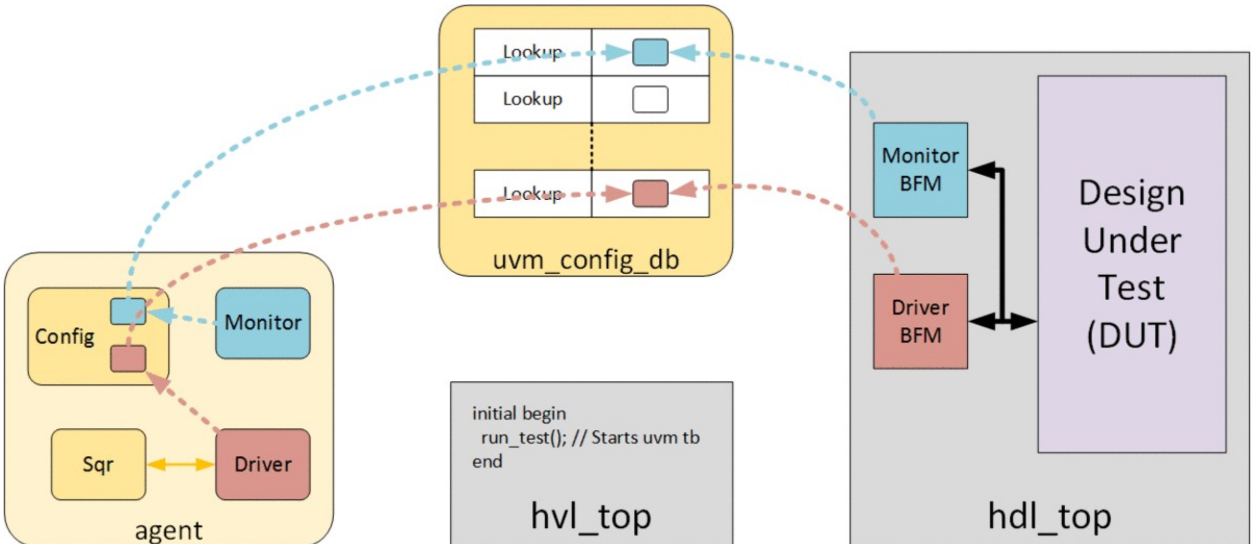
Connecting the Testbench to the DUT

Topic Overview

In this section of the UVM cookbook we consider the problem of connecting the UVM testbench to the RTL DUT.

The UVM testbench objects cannot make a direct connection to the DUT signals in order to drive or sample them. The connection between an agents driver and monitor component objects and the DUT is indirected through one or more BFM components that have static signal ports. These BFM components are implemented either as modules or interfaces and in order to complete the connection to a UVM monitor or driver component class we use a SystemVerilog virtual interface handle to reference the content of a static interface.

The UVM methodology uses the uvm_config_db utility to pass a virtual interface handle from a static testbench module to a UVM object class. Two examples of how to do this can be found in the section on using a virtual interface to implement BFMs.



Using the uvm_config_db to pass virtual interface handles from hdl_top to an agent

Overview showing how concrete BFM interfaces in the hdl_top are referenced from the agent monitor and driver UVM objects via virtual interface handles in the uvm_config_db

In most examples in the UVM cookbook we use the "Dual Top" methodology. What this means is that the DUT and the BFMs instantiated in one static module hierarchy, the top of which is usually referred to as the "hdl_top", and that the part of the testbench that is responsible for starting up the UVM test is in a separate module which is referred to as the "hvl_top". This separation helps when there are separate teams working on the design and verification sides of the project, and it also helps when moving to testbench acceleration since at that point the contents of the hdl_top will be synthesiseable.

We also consider an alternative way of connecting from a SystemVerilog class object to a static BFM using an abstract-concrete class pair. This approach uses polymorphism and a class handle to communicate with a BFM using an API.

Finally we discuss the handling of parameters in DUT-TB connections, where parameters are used to specify the width of signal fields in a protocol interface.

Interfaces and Virtual Interfaces

SystemVerilog Interfaces

The SystemVerilog interface provides a convenient means of organising related signals into a container in order to simplify connections between modules. An interface can contain:

- Port declarations
- Signal and variable declarations
- Any SystemVerilog code except a module instantiation
- Modport
- Other interfaces

An interface may be declared with or without ports. If it is declared with ports then those ports need to be assigned to signals when the interface is instantiated.

All signals declared as ports of the interface or within the interface can be passed between modules through one interface entity. A module can have interface ports and may mix these with other types of signal ports. Signals within an interface can be referenced or be assigned using a hierarchical reference within the interface – i.e. interface_name.signal_name

Interfaces can include any SystemVerilog code except a module instantiation. This means that they can contain initial and always blocks, tasks, functions, SystemVerilog assertions (SVA), covergroups, and classes, all of which are useful for constructing verification components.

SystemVerilog interfaces can also contain modports, which allow the direction of the interface signals to be organised dependent on a useage viewpoint. For instance, a bus master would always drive an address and strobe signal, whilst a bus slave would always receive these signals. Although, from a methodology perspective this appears to be a good idea, there is an element of clumsiness in the language surrounding the modport construct for it not to be particularly useful in practice.

The Virtual Interface

In SystemVerilog a class cannot make a reference to a signal without being declared within a module or interface scope where those signals are defined. For the purposes of developing reusable testbenches this is very restrictive.

However, SystemVerilog classes can reference signals within an interface via a virtual interface handle. This allows a class to either assign or sample values of signals inside an interface, or to call tasks or functions within an interface.

In order to reference an interface from within a class it must be declared as a virtual interface:

```
class protocol_driver extends uvm_driver #(protocol_seq_item);

virtual protocol_interface vif; // Virtual interface declaration

....

task run_phase(uvm_phase phase);

  @(posedge vif.clock); // Accessing the clock signal via the virtual interface handle
```

```
vif.send_packet(....); // Calling a BFM method via the virtual interface handle
```

If a virtual interface is not assigned, it has a default value of null and any attempt by the class to access something inside it will result in a null pointer exception in the simulator. For the virtual interface handle to become valid, it must be assigned to a concrete static interface instance in the testbench module part of the testbench. There are various ways in which this can be done SystemVerilog, but for the UVM the recommended and most scaleable way is to use the uvm_config_db to make this assignment.

The UVM Configuration Database (uvm_config_db)

The uvm_config_db

The uvm_config_db is a UVM utility class that is used to pass configuration data objects between component objects in a UVM testbench. Any call to the uvm_config_db is parameterised with the type of the data object that is stored or retrieved. To store an object in the uvm_config_db, its set() method is used, and to retrieve an object from the uvm_config_db, its get() method is used. Both these methods have arguments that assign the object; associate the object with a lookup key; and define which components within the UVM testbench hierarchy can reference the object.

For more information on the general use of the uvm_config_db see the following UVM cookbook page.

Connecting a UVM testbench to a DUT using the uvm_config_db

To pass a virtual interface handle from the static module based part of the testbench (hdl_top), to a UVM component, use the uvm_config_db as follows:

Step 1: In the HDL part of the testbench assign the static interface to a virtual interface entry in the uvm_config_db:

```
sfr_if SFR_IF(); // Declaration of SFR interface -This will be
connected to modules
initial begin
  uvm_config_db #(virtual sfr_if)::set(null, “uvm_test_top”, “SFR”,
SFR_IF);
end
```

Note the following:

- The uvm_config_db is parameterised with the type ***virtual sfr_if***
- The first argument of the set() method is context, intended to be assigned a UVM component object handle; in this case since we are in the HDL part of the testbench, a null object handle is assigned.
- The second argument of the set() method is a string used to identify the UVM component instance name(s) within the UVM testbench component hierarchy that may access the data object. This is “uvm_test_top” here to restrict access to the top level UVM test object. It could have been assigned a wildcard such as “*”, which means that all components in the UVM testbench could access it, but this may not be helpful, and carrys a potential lookup overhead in the get() process.
- The third argument of the set() method is a string, intended as the lookup name, i.e. a string that can be used to uniquely identify the virtual interface from within the uvm_config_db.
- The final argument of the set() is the static interface assigned to the virtual interface handle entry that is created within the uvm_config_db.

Step 2: In a UVM test component assign the virtual interface handle from the uvm_config_db entry to the virtual interface handle inside a configuration object:

```
// Virtual interface handle declared inside an agent configuration
object
class sfr_config extends uvm_object;
...
virtual sfr_if sfr_vif;
...
endclass

class test extends uvm_test;
...

function void build_phase(uvm_phase phase);
  sfr_config cfg = sfr_config::type_id::create("cfg");

  // Assigning the SFR virtual interface handle via the uvm_config_db
#(...)::get() method
  if(!uvm_config_db #(virtual sfr_if)::get(this, "", "SFR",
cfg.sfr_vif)) begin
    `uvm_error("VIF_GET", "Unable to find the SFR virtual interface in
the uvm_config_db", UVM_LOW)
  end
```

Note the following:

- The uvm_config_db::get() method is a function that returns a bit value to indicate whether the object retrieval was been successful or not; this is tested to ensure that the testbench does not proceed if the lookup fails.
- The uvm_config_db::get() method is parameterised with the virtual interface type so that the right type of object is retrieved from the database.
- The first argument of the get() method, context, is passed the handle for the UVM component in which the call is being made – this
- The second argument, instance_name, is passed an empty string, “”, this means that only the path string for the component is applied based on the component path string derived from the component handle passed in context (i.e. “uvm_test_top”).
- The third argument is the uvm_config_db lookup string, this should match the lookup string assigned in the HDL module, in this case “SFR”.
- The fourth argument is the virtual interface handle. In this case the handle is inside a uvm_config_object that will be passed to an agent.

If you are reusing a verification component, then all you need to know is the name of its interface and how to assign it to the UVM verification components configuration object. You can then apply the above recipe to make the virtual interface assignments. If you have to create a new verification component then you need to be aware of some implementation considerations.

Virtual Interface BFMs

In order to make verification components reusable between testbenches they are organised as uvm_agents with an associated signal interface. These are also referred to as UVCs (Universal Verification Components). Inside the uvm_agent, there are two types of uvm_components that interact with the virtual interface. The driver component is responsible for the active part of the testbench, converting information contained within sequence_items into interface activity which involves driving and sampling signal values via a virtual interface handle. The monitor is responsible for the passive side of the testbench, and will sample interface activity via a virtual interface handle to create sequence_items that represent interface events.

When designing verification components, there are various ways in which the driver and the monitor can be implemented:

- The interface is a signal container and all signal driving and sampling takes place directly in the UVM driver and monitor classes.
- The interface has methods to either make interface transfers or to sample interface transfers and these are called from the UVM driver and monitor classes.
- A mixture of the two

The simplest approach is to keep all signal accesses within the UVM driver and monitor components and leave the interface as a signal bundle. If the interface is fairly simple, and simulation is the only intended useage for the verification component, then this approach is acceptable.

However, if the potential use models for the verification component are likely to include hardware-assisted acceleration platform using emulation or an FPGA prototyping platform, then the driver and monitor implementations have to be changed so that they are based on calling methods in the interface, rather than direct signal access. This enables the interface to be made synthesisable at some point. In the first instance, an interface might be implemented with methods that use non-synthesisable behavioural code that will only work in a simulator, but in a later second pass these methods can be re-implemented to use synthesisable code without needing to change the UVM driver or monitor code.

Ultimately, the most scalable approach is to determine which parts of the protocol abstraction are best implemented as UVM component code using the strengths of object oriented programming and which are best implemented as a synthesisable task called in the interface. An example of this would be an Ethernet protocol where assembling all the bits in a packet might be best done in the class world, so that sending the packet can be delegated to a relatively simple synthesisable task in the interface.

To illustrate how different implementations might work out in practice, let us consider two alternative ways of implementing the same verification component. In the first example, the interface is a simple signal container and the driver and monitor components reference a virtual interface to drive and monitor interface transfers. In the second example, the interface has tasks to drive and monitor interface transfers and the UVM driver and monitor components reference a virtual interface to call these tasks indirectly.

Both examples separate out the HDL domain of the testbench and the HVL (Hardware Verification Language) or UVM domain of the testbench into two separate modules. This is referred to as the two or dual top testbench architecture.

For testbenches that may eventually need to be co-simulated with an emulator or a FPGA prototype dual domain partitioning [1] is necessary to group all of the synthesisable code into a separate HDL module hierarchy that can be compiled for a hardware platform. All the non-synthesisable UVM start up code resides in the HVL module. For other testbenches, the dual top approach provides a convenient means of separating out concerns, enables design teams to make changes to the HDL domain without impacting the verification environment, and it enables verification teams to make changes to the HVL domain without impacting the design team.

Example 1 - Two top, simulation only version

In the first example, of an agent only intended for simulation, the protocol interface is declared as a signal bundle and is instantiated in the hdl_top testbench module.

```
interface sfr_if(input clk, input reset);

  logic[7:0] address;
  logic[7:0] write_data;
  logic[7:0] read_data;
  logic we;
  logic re;

endinterface: sfr_if
```

In the driver class of the agent, the conversion of the transaction (sequence_item) content to the clock cycle based stimulus pin wiggles takes place inside the run_phase() task, with the signals either being driven or sampled via the virtual interface handle.

```
task sfr_driver::run_phase(uvm_phase phase);
  sfr_seq_item item;

  forever begin
    seq_item_port.get_next_item(item);
    if(SFR.reset == 1) begin
      SFR.re <= 0;
      SFR.we <= 0;
      SFR.address <= 0;
      SFR.write_data <= 0;
      wait(SFR.reset == 0);
    end
    else begin
      @(posedge SFR.clk);
      SFR.address = item.address;
      SFR.we <= item.we;
      SFR.write_data <= item.write_data;
      SFR.re <= item.re;
      @(posedge SFR.clk);
      if(SFR.re == 1) begin
        item.read_data = SFR.read_data;
        SFR.re <= 0;
      end
      SFR.we <= 0;
    end
    seq_item_port.item_done();
  end

endtask: run_phase
```

In the monitor class of the agent, the monitor samples the interface signals via the virtual interface handle during its run_phase() task, and publishes the corresponding analysis transactions (sequence_items) when it determines that a transaction has completed.

```
task sfr_monitor::run_phase(uvm_phase phase);
  sfr_seq_item item;

  forever begin
    @(posedge SFR.clk);
    if((SFR.we == 1) || (SFR.re == 1)) begin
      item = sfr_seq_item::type_id::create("item");
      item.we = SFR.we;
      item.re = SFR.re;
      item.address = SFR.address;
      item.write_data = SFR.write_data;
      item.read_data = SFR.read_data;
      ap.write(item);
    end
  end

endtask: run_phase
```

The testbench has two top level modules, namely the hdl_top, which contains the interface and the DUT, and the hvl_top with the initial block for starting the UVM testbench. The following two code snippets represent the important code from these two modules.

'''The hdl_top module'''

```
module hdl_top;

import uvm_pkg::*;

sfr_if SFR(.clk(clk), .reset(reset));

sfr_dut dut (.clk(clk),
             .reset(reset),
             .address(SFR.address),
             .write_data(SFR.write_data),
             .we(SFR.we),
             .re(SFR.re),
             .read_data(SFR.read_data));

initial begin
  uvm_config_db #(virtual sfr_if)::set(null, "uvm_test_top", "SFR",
SFR);
end

endmodule
```

'''The hvl_top module'''

```
module hvl_top;

import uvm_pkg::*;
import sfr_test_pkg::*;

initial begin
  run_test();
end

endmodule: hvl_top
```

In summary, this example uses only one interface, and all of the signal processing takes place in the class-based UVM testbench domain.

Example 2 - Two top, emulation friendly version

In the second example, there are two separate interfaces, one for driving stimulus (driver_bfm) and one for monitoring the interface (monitor_bfm). For emulation friendliness, the agents signal level driver and monitoring functionality has been moved to the interface as a method, and the UVM agent driver and monitor objects make the method call via the appropriate virtual interface handle [1].

'''The Driver'''

```
task sfr_driver::run_phase(uvm_phase phase);
  sfr_seq_item item;

  forever begin
    seq_item_port.get_next_item(item);
    SFR.execute(item);
    seq_item_port.item_done();
  end
```

'''The Driver BFM'''

```
  task execute(sfr_seq_item item);
    if(reset == 1) begin
      wait(reset == 0);
    end
    else begin
      @(posedge clk);
      address = item.address;
      we <= item.we;
      write_data <= item.write_data;
      re <= item.re;
      @(posedge clk);
      if(re == 1) begin
        item.read_data = read_data;
        re <= 0;
      end
      we <= 0;
    end
```

```
  endtask: execute
```

'''The Monitor'''

```
task sfr_monitor::run_phase(uvm_phase phase);
  sfr_seq_item item;

  forever begin
    item = sfr_seq_item::type_id::create("item");
    SFR.monitor(item);
    ap.write(item);
  end

endtask: run_phase
```

'''The Monitor BFM'''

```
  task monitor(sfr_seq_item item);
    @(posedge clk);
    if((we == 1) || (re == 1)) begin
      item.we = we;
      item.re = re;
      item.address = address;
      item.write_data = write_data;
      item.read_data = read_data;
    end
  endtask: monitor
```

If you compare the two examples, you should note that the functionality is essentially the same, and that the only difference is the delegation of the signal handling to methods in the bfm interfaces. Although, the code in the interfaces is not synthesisable, the API call required to either execute a transaction or to monitor a bus transfer is in place, ready for the bfm interfaces to be replaced with a synthesisable implementation.

In the hdl_top testbench module there are now two interfaces (master_bfm and monitor_bfm) and this requires two virtual interfaces to be set in the uvm_config_db:

```
module hdl_top;

import uvm_pkg::*;

sfr_master_bfm SFR_MASTER(.clk(clk),
                          .reset(reset),
                          .address(address),
                          .write_data(write_data),
                          .we(we),
                          .re(re),
                          .read_data(read_data));

sfr_monitor_bfm SFR_MONITOR(.clk(clk),
                            .reset(reset),
                            .address(address),
```

```
                            .write_data(write_data),
                            .we(we),
                            .re(re),
                            .read_data(read_data));

sfr_dut dut (.clk(clk),
             .reset(reset),
             .address(address),
             .write_data(write_data),
             .we(we),
             .re(re),
             .read_data(read_data));

initial begin
  uvm_config_db #(virtual sfr_master_bfm)::set(null, "uvm_test_top",
"SFR_MASTER", SFR_MASTER);
  uvm_config_db #(virtual sfr_monitor_bfm)::set(null, "uvm_test_top",
"SFR_MONITOR", SFR_MONITOR);
end

endmodule
```

The hvl_top testbench module is exactly the same as in example 1.

In summary, the emulation friendly version of the example uses two separate interfaces, and the signal driving and monitoring functionality is initiated via API calls in those interfaces. This example can be used for simulation purposes and provides a bridge to an implementation where the bfm interfaces are replaced with an implementation that would run on an emulator or a hardware prototyping platform.

Most other examples in the UVM cookbook will follow the emulation friendly approach. The exceptions will be those that have been simplified to avoid the use of agents with interfaces in order to illustrate a point.

Handling Parameterization

Parameters are commonly used to configure design IP and interfaces. From the perspective of VIP, parameters usually affect the width of bus fields or the number of channels or lanes in use.

SystemVerilog interfaces may be parameterised and, when they are, their virtual interface handles also need to be parameterised. This impaacts the UVM VIP code since the parameter values of the virtual interface inside the monitor and driver of the agent need to match those of the static interface in the hdl_top testbench module. Assigning different parameter values to an interface effectively creates a new type.

Handling parameters in RTL is well understood. It usually requires top level parameters to be passed down the design hierarchy, with some possible modification. Parameter can be handled in the same way in class libraries, but this is often inconvenient since in the worst case it means that every sequence item has to be parameterised. If there are several parameterised interfaces of the same type, but with different parameters, then there is scope for error. There are two main strategies that are used to cope with parameters in VIP interfaces:

- Use the maximum possible value of the parameter and only connect the wires and channels used
- Use a parameter package to manage the parameters and typedef parameterised classes

In the example that goes with this article, we use a mixture of the two.

The testbench parameter package

Using a testbench parameter package allows you to manage your testbench parameters. Keeping all the parameter definitions in one package allows you to ensure that there is only one file that needs to be edited should parameters change.

The code example below shows the typical content of a testbench parameter package. The package needs to import all the VIP agent packages so that specialised versions of the parameterised classes can be typedef'ed. For each interface, a class should be defined that contains the parameter values defined as localparams. This class then provides a scoped container for those parameter values. Then the parameterised classes that the user has to deal are defined as specialised typedefs using the class scope. Finally, the typedefs for the specialised virtual interface handles are declared.

```
package tb_params_pkg;

// Import the VIP package to get access to all the VIP class types
import sfr_agent_pkg::*;

// Class declaration with localparams - this creates a defined
namespace
// and is useful for handling multiple VIPs of the same type
class SFR;

  localparam sfr_addr_width = 16;
  localparam sfr_data_width = 32;

endclass


// typedefs for those parts of the agent code exposed to the user,
using the typedefs
```

```
// avoids having to type in the parameters
typedef sfr_config_object #(SFR::sfr_addr_width, SFR::sfr_data_width)
SFR_cfg_t;
typedef sfr_agent #(SFR::sfr_addr_width, SFR::sfr_data_width)
SFR_agent_t;

// Declaration of virtual interface handles
typedef virtual sfr_monitor_bfm #(SFR::sfr_addr_width,
SFR::sfr_data_width) SFR_monitor_bfm_t;
typedef virtual sfr_master_bfm #(SFR::sfr_addr_width,
SFR::sfr_data_width) SFR_master_bfm_t;

endpackage
```

This example only shows the content for one VIP, but when there are multiple VIPs with parameters they should all be grouped in the same testbench parameter package.

Using parameterised interfaces with the uvm_config_db

If you have parameterised interaces in your hdl_top testbench module you will need to use the uvm_config_db #()::set() method with the same parameters. There are two ways to do this, either using the parameters from the tb_params_pkg, or by using a specialised typedef for the virtual interface as defined in the package:

```
import uvm_pkg::*;
import tb_params_pkg::*;  // Import of the tb_params_pkg to access
typedefs

logic clk;
logic reset;

wire[SFR::sfr_addr_width-1:0] address;
wire[SFR::sfr_data_width-1:0] write_data;
wire[SFR::sfr_data_width-1:0] read_data;
wire we;
wire re;

// Parameterised version of the sfr_master_bfm using the SFR scope to
define the parameters:
sfr_master_bfm #(.ADDR_WIDTH(SFR::sfr_addr_width),
                 .DATA_WIDTH(SFR::sfr_data_width))
SFR_MASTER(.clk(clk),

.reset(reset),

.address(address),

.write_data(write_data),

.read_data(read_data),
```

```
                                                                    .re(re),
                                                                    .we(we));

sfr_monitor_bfm #(.ADDR_WIDTH(SFR::sfr_addr_width),
                  .DATA_WIDTH(SFR::sfr_data_width))
SFR_MONITOR(.clk(clk),

.reset(reset),

.address(address),

.write_data(write_data),

.read_data(read_data),

.re(re),

.we(we));

initial begin
  // Using a parameterised virtual interface handle:
  uvm_config_db #(virtual sfr_master_bfm
#(.ADDR_WIDTH(SFR::sfr_addr_width),

.DATA_WIDTH(SFR::sfr_data_width)))::set(null, "uvm_test_top",
"SFR_MASTER", SFR_MASTER);

  // Using the virtual interface type declared in the tb_params_pkg:
  uvm_config_db #(SFR_monitor_bfm_t)::set(null, "uvm_test_top",
"SFR_MONITOR", SFR_MONITOR);
end
```

In the UVM test, the virtual interface handle typedef from the tb_params_pkg is used to parameterise the uvm_config_db get() call:

```
class sfr_test extends uvm_component;


sfr_env_config env_cfg;
SFR_cfg_t sfr_agent_cfg; // Typedef from tb_params_pkg

sfr_env env;

extern function void build_phase(uvm_phase phase);
extern task run_phase(uvm_phase phase);

endclass: sfr_test
```

```
function void sfr_test::build_phase(uvm_phase phase);
  env_cfg = sfr_env_config::type_id::create("env_cfg");
  sfr_agent_cfg = SFR_cfg_t:: type_id::create("sfr_agent_cfg");
  //
  // Getting the parameterised virtual interface handles from the
uvm_config_db
  //
  if(!uvm_config_db #(SFR_master_bfm_t)::get(this, "", "SFR_MASTER",
sfr_agent_cfg.SFR_MASTER)) begin
    `uvm_error("BUILD_PHASE", "Unable to find virtual interface
sfr_master_bfm in the uvm_config_db")
  end
  if(!uvm_config_db #(SFR_monitor_bfm_t)::get(this, "", "SFR_MONITOR",
sfr_agent_cfg.SFR_MONITOR)) begin
    `uvm_error("BUILD_PHASE", "Unable to find virtual interface
sfr_master_bfm in the uvm_config_db")
  end
  sfr_agent_cfg.is_active = 1;
  env_cfg.sfr_agent_cfg = sfr_agent_cfg;
  env = sfr_env::type_id::create("env", this);
  env.cfg = env_cfg;
endfunction: build_phase
```

In the env_config and the env classes, the typedefs for the specialised version of the sfr_agent_config and sfr_agent classes are used to pass the parameter values down through the class hierarchy. Using the tb_params_pkg definitions ensures that all the class hierarchy parameterisation is consistent.

```
// The sfr_env_config class:
class sfr_env_config extends uvm_object;

SFR_cfg_t sfr_agent_cfg;

endclass: sfr_env_config

// The sfr_env
class sfr_env extends uvm_component;

sfr_env_config cfg;
sfr_scoreboard sb;
SFR_agent_t agent; // Using the SFR_agent_t typedef from tb_params_pkg

extern function void build_phase(uvm_phase phase);
extern function void connect_phase(uvm_phase phase);

endclass: sfr_env

function void sfr_env::build_phase(uvm_phase phase);
  if(cfg == null) begin
```

```
        if(!uvm_config_db #(sfr_env_config)::get(this, "", "CFG", cfg))
begin
            `uvm_error("BUILD_PHASE", "Unable to find environment configuration
object in the uvm_config_db")
        end
    end
    sb = sfr_scoreboard::type_id::create("sb", this); agent =
    SFR_agent_t::type_id::create("agent", this); agent.cfg = cfg.sfr_agent_cfg;
endfunction: build_phase

// And the sfr_agent - to show how the parameters are passed through to
  the driver.
// Note that the parameters are defined more generically here to ensure
  that the agent can be reused
//
class sfr_agent #(SFR_ADDR_WIDTH = 8, SFR_DATA_WIDTH = 8) extends
uvm_component;

typedef sfr_agent #(SFR_ADDR_WIDTH, SFR_DATA_WIDTH) this_t;

`uvm_component_param_utils(this_t) uvm_analysis_port

#(sfr_seq_item) ap; uvm_sequencer #(sfr_seq_item)

sequencer;

sfr_driver #(SFR_ADDR_WIDTH, SFR_DATA_WIDTH) driver; sfr_monitor
#(SFR_ADDR_WIDTH, SFR_DATA_WIDTH) monitor;

sfr_config_object #(SFR_ADDR_WIDTH, SFR_DATA_WIDTH) cfg;

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

extern function void build_phase(uvm_phase phase);
extern function void connect_phase(uvm_phase phase);

endclass: sfr_agent

function void sfr_agent::build_phase(uvm_phase phase);
    if(cfg == null) begin
        if(!uvm_config_db #(sfr_config_object #(SFR_ADDR_WIDTH,
SFR_DATA_WIDTH))::get(this, "", "SFR_CFG", cfg)) begin
            `uvm_error("BUILD_PHASE", "Unable to find sfr agent config object in the uvm_config_db")
```

```
    end
  end
  ap = new("ap", this);
  monitor = sfr_monitor #(SFR_ADDR_WIDTH,
SFR_DATA_WIDTH)::type_id::create("monitor", this);
  if(cfg.is_active == 1) begin
    driver = sfr_driver #(SFR_ADDR_WIDTH,
SFR_DATA_WIDTH)::type_id::create("driver", this);
    sequencer = uvm_sequencer
#(sfr_seq_item)::type_id::create("sequencer", this);
  end
endfunction: build_phase
```

Finally, in order to ensure that any sequences can be written without having to deal with parameters, we have taken the "max width" approach to the sequence_item item definition. The maximum width of the address and data buses is 32 bits, so we have used this to size the the address and data fields in the sequence item. Any unused bits will be ignored.

```
class sfr_seq_item extends uvm_sequence_item;

`uvm_object_utils(sfr_seq_item)

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

rand bit[31:0] address;
rand bit[31:0] write_data;
rand bit we;
rand bit re;

bit[31:0] read_data;

endclass: sfr_seq_item
```

The reason that the UVM classes have to take on parameters is that the virtual interface handles have to be declared and assigned using parameters, an alternative solution to this problem is to use the abstract-concrete class implementation approach.

Abstract-Concrete Class Connections

Note: This approach to DUT-TB connections is a simulation only technique

An alternative to using a virtual interface handle for DUT to UVM testbench connections is to use an abstract concrete class pair. The abstract version of the class defines templates for all the methods that are available in the BFM. The concrete version of the class is declared in the BFM and it extends the abstract version, providing an implementation of the methods. The handle of the concrete class is assigned to the abstract handle in the agent classes via the uvm_config_db. The properties of class object polymorphism allow the abstract class handle to interact with the BFM using the method implementations in the concrete version of the class.

If a class is defined inside an interface or a module, it can reference any of the variables within the interface or module scope. This can be used to simplify the handling of parameters.

The following code snippets show the Abstract and Concrete versions of the sfr_master class:

```
// The abstract class for the SFR master driver/BFM
class sfr_master_abstract;

virtual task execute(sfr_seq_item item);

endtask

endclass

// The implementation for the SFR master BFM, containing the concrete
implementation of the sfr_master
module sfr_master_bfm  #(ADDR_WIDTH = 8,
                         DATA_WIDTH = 8)
                        (input clk,
                         input reset,
                         output logic[ADDR_WIDTH-1:0] address,
                         output logic[DATA_WIDTH-1:0] write_data,
                         input logic[DATA_WIDTH-1:0] read_data,
                         output logic we,
                         output logic re);

  import uvm_pkg::*;
  import sfr_agent_pkg::*;

  always @(reset or posedge clk) begin
    if(reset == 1) begin
      re <= 0;
      we <= 0;
      address <= 0;
      write_data <= 0;
    end
  end

// Definition of the concrete version of the sfr_master class:
```

```
class sfr_master_concrete extends sfr_master_abstract;

  task execute(sfr_seq_item item);
    if(reset == 1) begin
      wait(reset == 0);
    end
    else begin
      @(posedge clk);
      address = item.address;
      we <= item.we;
      write_data <= item.write_data;
      re <= item.re;
      @(posedge clk);
      if(re == 1) begin
        item.read_data = read_data;
        re <= 0;
      end
      we <= 0;
    end
  endtask: execute
endclass

// Declaration of the SFR_MASTER class handle
sfr_master_concrete SFR_MASTER;

// Initial block, constructing the SFR_MASTER object and assigning the
handle into the uvm_config_db
// Note the use of the sfr_master_abstract handle as the uvm_config_db
parameter
initial begin
  SFR_MASTER = new();
  uvm_config_db #(sfr_master_abstract)::set(null, "uvm_test_top",
"SFR_MASTER", SFR_MASTER);
end

endmodule: sfr_master_bfm
```

Note that in the above code, the BFM has been changed to a module from an interface, this is another freedom that using the abstract-concrete pattern enables. Modules allow the BFM to have hierarchy.

Inside the SFR agent classes, any reference to the BFM is via handles of the sfr_master_abstract type:

```
class sfr_driver extends uvm_driver #(sfr_seq_item);

// Abstract version of the sfr_master class:
sfr_master_abstract SFR;

extern task run_phase(uvm_phase phase);
```

```
endclass: sfr_driver

task sfr_driver::run_phase(uvm_phase phase);
  sfr_seq_item item;

  forever begin
    seq_item_port.get_next_item(item);
    SFR.execute(item);
    seq_item_port.item_done();
  end

endtask: run_phase
```

The handle to the sfr_master_abstract object is passed via the agent config object via the uvm_config_db, note that in the following code snippet, the sfr_monitor uses the same connection pattern:

```
class sfr_test extends uvm_component;



sfr_env_config env_cfg;
sfr_config_object sfr_agent_cfg;

sfr_env env;

extern function void build_phase(uvm_phase phase);
extern task run_phase(uvm_phase phase);

endclass: sfr_test

function void sfr_test::build_phase(uvm_phase phase);
  env_cfg = sfr_env_config::type_id::create("env_cfg");
  sfr_agent_cfg = sfr_config_object:: type_id::create("sfr_agent_cfg");

  // Assigning the handle to the concrete implementations of the
sfr_master and sfr_monitor classes to their abstract counterparts
  if(!uvm_config_db #(sfr_master_abstract)::get(this, "", "SFR_MASTER",
 sfr_agent_cfg.SFR_MASTER)) begin
    `uvm_error("BUILD_PHASE", "Unable to find virtual interface
sfr_master_bfm in the uvm_config_db")
  end
  if(!uvm_config_db #(sfr_monitor_abstract)::get(this, "",
"SFR_MONITOR", sfr_agent_cfg.SFR_MONITOR)) begin
    `uvm_error("BUILD_PHASE", "Unable to find virtual interface
sfr_master_bfm in the uvm_config_db")
  end

  sfr_agent_cfg.is_active = 1;
  env_cfg.sfr_agent_cfg = sfr_agent_cfg;
```

```
  env = sfr_env::type_id::create("env", this);
  env.cfg = env_cfg;
endfunction: build_phase
```

When the abstract-concrete class connection pattern is used, the agent does not have to be parameterised, since class handles are used to make the connection.

In summary, the abstract concrete class approach works by passing a class object handle from the hdl_top part of the testbench to the UVM testbench.

Parameterized Tests

Introduction

SystemVerilog provides a number of ways to pass changeable values through different code structures. Some changeable values must be fixed at elaboration time and others can be changed at run-time after starting a simulation. Changeable values fixed at elaboration time are represented using either a SystemVerilog parameter or `define macro. The use of `define macros causes complications in cases of multiple instances of a module or interface where each instance needs a different changeable value. SystemVerilog parameters are more fitting and the preferred mechanism to pass changeable "type" values as well as to specify bit vector sizes such as data and address widths.

The values of "size" parameters used in the design to control widths are generally required also in the testbench. The remainder of this article illustrates a method to share parameters through parameterized UVM tests inside the dual top testbench architecture.

DVCon Papers

The information in this article is drawn from a conference paper with Xilinx at DVCon 2011 [1], and a subsequent updated paper at DVCon 2016 [2] fit to the dual top testbench architecture. 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 UVM factory. The UVM actually has two factories - one string-based and one type-based. These param_utils macros only perform registration with the type-based factory.

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;
 // <-- Default null string used for unspecified 2nd string parameter

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 *`uvm_component_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.

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()*. The *run_test()* call 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 UVM component is not registered with the string-based factory by default (per above example), you will need to implement a string-based registration for your top-level test classes so that they can be instantiated by *run_test()*. To accomplish this, you must manually code the actions that the *`uvm_component_param_utils* macro performs.

To perform 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; // <-- Unique string specified as 2nd parameter
```

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

```
module hvl_top();
  parameter  DATA_WIDTH = 4;

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

  initial begin
    run_test("basic_test1");

endmodule
```

Note: Instead of a name like "basic_test1" above, 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.

Sharing Parameters with Dual Tops

As shown above the hvl_top module contains a parameter DATA_WIDTH defined in the module. Looking at the hdl_top, it also has a parameter called DATA_WIDTH.

```
module hvl_top();
  parameter  DATA_WIDTH = 4;

  //Pin and BFM interface instantiations using DATA_WIDTH

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

  ...
```

```
endmodule
```

Hence if this parameter changes, updates are required in two locations. This can be overcome by using a shared parameter package to define the parameters in a single package which is then imported into both the hvl_top and hdl_top.

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, such as adding a new parameter, often 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.

Declaration	`define params_declare #(int BUS_WIDTH = 16, int ADDR_WIDTH = 8)
Instantiation / Mapping	`define params_map #(.BUS_WIDTH(BUS_WIDTH), .ADDR_WIDTH(ADDR_WIDTH))
String Value	`define params_string $sformat("#(%1d, %1d)", BUS_WIDTH, ADDR_WIDTH)

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.

References

[1] https://verificationacademy.com/resource/6648
[2] http://events.dvcon.org/2016/proceedings/papers/08_4.pdf

Configuring a Test Environment

Testbench Configuration

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 (i.e. reconfigured).

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, other constraint expression values and coverage bin values. These values can be represented by SystemVerilog variables, which can be set (and changed) at run-time, or SystemVerilog parameters, which must be set at elaboration time. Because of the flexibility they offer, variables organized into configuration objects and accessed using the uvm_config_db API should always be used where possible.

On the other hand, aspects like bus widths must be fixed at elaboration time, and so cannot be implemented via dynamic configuration objects. There are a number of articles on handling static parameters in UVM:

- The Parameterized Tests article shows how to use parameterized tests with the UVM factory.
- The Parameters Package article shows how to centralize the parameters shared between HDL/DUT and HVL/TB domains.
- The Parameters and Reuse article shows how to pass large sets of parameters down through the uvm_component hierarchy.

Configuration Objects

Configuration objects are an efficient, reusable means for organizing configuration variables. In a typical testbench there will generally be several configuration objects, each tied to a component. A configuration object is created as a subclass of uvm_object to encapsulate all related configuration variables for a given branch of the testbench structural hierarchy. There may also be a single, additional configuration object to hold global configuration variables. Each of the configuration variables within a configuration object may be declared as rand and consequently the configuration object may be randomized.

The UVM configuration database is effective in handling the scope and storage of user defined configuration objects. Below is the code for a typical agent configuration object. It has virtual interfaces to point to the driver and monitor BFM interfaces associated with the agent, and a number of variables to describe and control these BFMs.

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

// Configuration Parameters
  virtual wb_m_bus_driver_bfm  wb_drv_bfm;  // virtual driver BFM
  virtual wb_bus_monitor_bfm   wb_mon_bfm;  // virtual monitor BFM

  int m_wb_id;                            // Wishbone bus ID
  int m_wb_master_id;                     // Wishbone bus master id for
```

```
wishone 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 utilizes a configuration object should perform the following steps:

- Retrieve its configuration if its config object handle is null (i.e. not already set externally)
- Create its internal structure and define its behavior based on the configuration
- Configure its child (sub-)components

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 variables 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 wb_m_bus_driver_bfm)::get(this, "",
"WB_DRV_BFM", wb_config_0.wb_drv_bfm))
       `uvm_fatal(...)
    if (!uvm_config_db #(virtual wb_bus_monitor_bfm)::get(this, "",
"WB_MON_BFM", wb_config_0.wb_mon_bfm))
```

```
      `uvm_fatal(...)

    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);
    set_wishbone_config_params();
    ...
  endfunction
  ...

endclass
```

Notice above that `uvm_fatal() is used if a virtual interface is not found in the UVM configuration database. This will stop the test immediately with the given message passed to the `uvm_fatal() call. Optionally, these `uvm_fatal() messages could be converted to `uvm_error() messages to allow more of the testbench to run before being halted.

The components that use a configuration object either have the configuration object passed to it directly or retrieve it by using uvm_config_db::get. In this example, the driver gets the virtual interface handle, ID, and verbosity from the configuration object. Note that BFMs are not uvm_components. Therefore, the driver and/or monitor proxy may also need to pass pertinent configuration data to the BFM through a function call. A single function call generally suffices to pass this data all at once, minimizing any call overhead.

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

  local virtual wb_m_bus_driver_bfm  m_bfm;  // Virtual Interface

  wb_config m_cfg;

  function void build_phase( uvm_phase phase );

    if (m_config == null)
      if( !uvm_config_db #( wb_config )::get( this , "" , "wb_config" ,
 m_cfg ) ) begin
        `uvm_fatal(...)
```

```
      end
    m_bfm = m_cfg.wb_drv_bfm; // set local virtual if property
    ...
  endfunction

  function void connect_phase( uvm_phase phase );
    super.connect_phase( phase );
    m_bfm.set_m_id(m_config.m_wb_master_id); //Set config value in BFM
  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.

Configuring DUT Connections

Setting up HDL-to-Testbench connections is a kind of configuration activity that is always necessary. A SystemVerilog module (typically the HDL side top level module or sometimes a finer level of module encapsulation must add virtual interfaces as applicable into the configuration space. On the HVL testbench side the test component retrieves the pertinent virtual interface handles from the UVM config database and applies them 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 wb_m_bus_driver_bfm)::get(this, "",
"WB_DRV_BFM", wb_config_0.wb_drv_bfm))
       `uvm_fatal(...)
    if (!uvm_config_db #(virtual wb_bus_monitor_bfm)::get(this, "",
"WB_MON_BFM", wb_config_0.wb_mon_bfm))
       `uvm_fatal(...)
    ...
    uvm_config_db #( wb_config )::set(this , "*", "wb_config",
wb_config_0, 0); // put in config
  endfunction
  ...
```

```
endclass
```

Configuring Sequences

A Configurable Sequence

The most generic way to configure a sequence is to use its full hierarchical name by default, but allow any other name to be set instead as required:

```
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
```

Consider a sequence called "initialization_sequence" running on the sequencer "uvm_test_top.env.sub_env.agent1.sequencer". The scope_name in the code above is set by default to "uvm_test_top.env.sub_env.agent1.sequencer.initialization_sequence".

The most common use case for sequence configuration takes the agent's configuration object set for the agent and its constituent components (sequencer, driver, monitor, ...).

```
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
```

With the sub_env used as context, the set() call and the default get() call in the configurable sequence will match and result in the sequence having access to the agent's configuration object.

Per Sequence Configuration

The default full hierarchical scope name of the configurable sequence class above can be readily used to uniquely identify multiple sequence instances and hence subject them to different individual configurations. All that is required is that the given instance names of the sequences are different.

For example, the environment class might look 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 class uses the given sequence name to do a config_db get() call, the normal sequence will pick up the configuration object with error injection disabled, while the error sequence will pick up the error configuration object.

Ignoring the Component Hierarchy Altogether

It is also possible to completely ignore the component hierarchy for sequence configuration. The advantage would be that one can in effect define behavioral scopes that are intended only for configuring sequences, and keep these behavioral scopes completely separate from the component hierarchy. The configurable sequence described above will work in this scenario as well.

For example in a virtual sequence one might have:

```
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 the added flexibility of this use model is that every sequence and component in the testbench must agree on the naming scheme, or at least be capable to deal with this kind of arbitrary naming scheme. As there is no longer guaranteed uniqueness of scope name, some reuse challenges may arise when transitioning from block to integration level testbench.

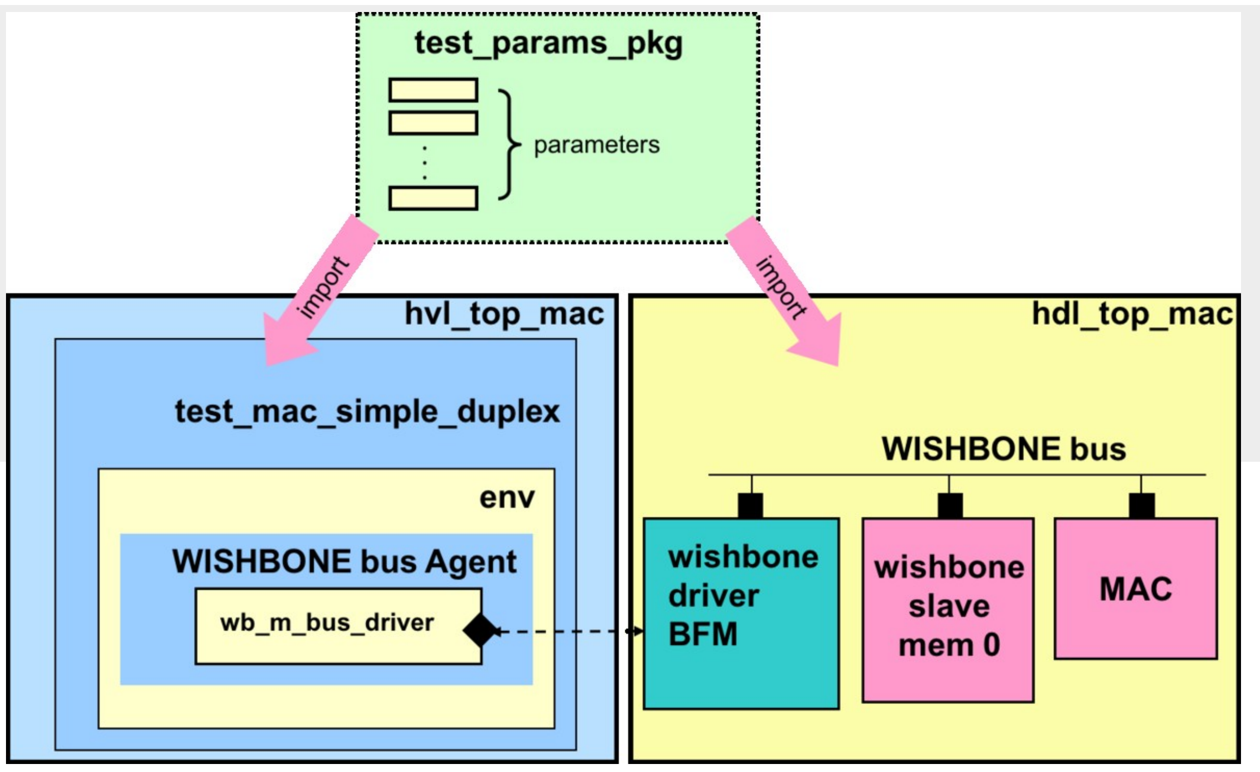
Using a Parameter Package

When a DUT or interface is parameterized, the parameter values are almost always used in the testbench as well. For this reason, these *common* parameters should not be specialized with direct literal values for your instance declarations. Define instead corresponding named parameters with associated values in a package to be shared by both the HDL/DUT side and testbench side of the environment. This greatly helps avoiding mistakes where a parameter value change would be made on one side but not the other, or where a test configuration parameter is some function of a DUT parameter and a miscalculation would result when making a change.

Note that this "shared package" need not be the place for all test parameters. Test parameters not applicable and used by the DUT can be specialized directly in the test. The shared parameters package should include only parameters shared between HDL/DUT and HVL/TB domains.

Example use of a Parameter Package

The WISHBONE example below involves two WISHBONE bus devices, slave memory and an Ethernet MAC (Media Access Controller). Parameters are put in a package *test_params_pkg* and used in instantiating the WISHBONE devices in the HDL top level module and in the test class on the testbench side.



```
  // 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 usage of parameters mem_slave_size and mem_slave_wb_id in the HDL 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_hdl module:

```
module hdl_top_mac;
  ...

  import test_params_pkg::*;

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

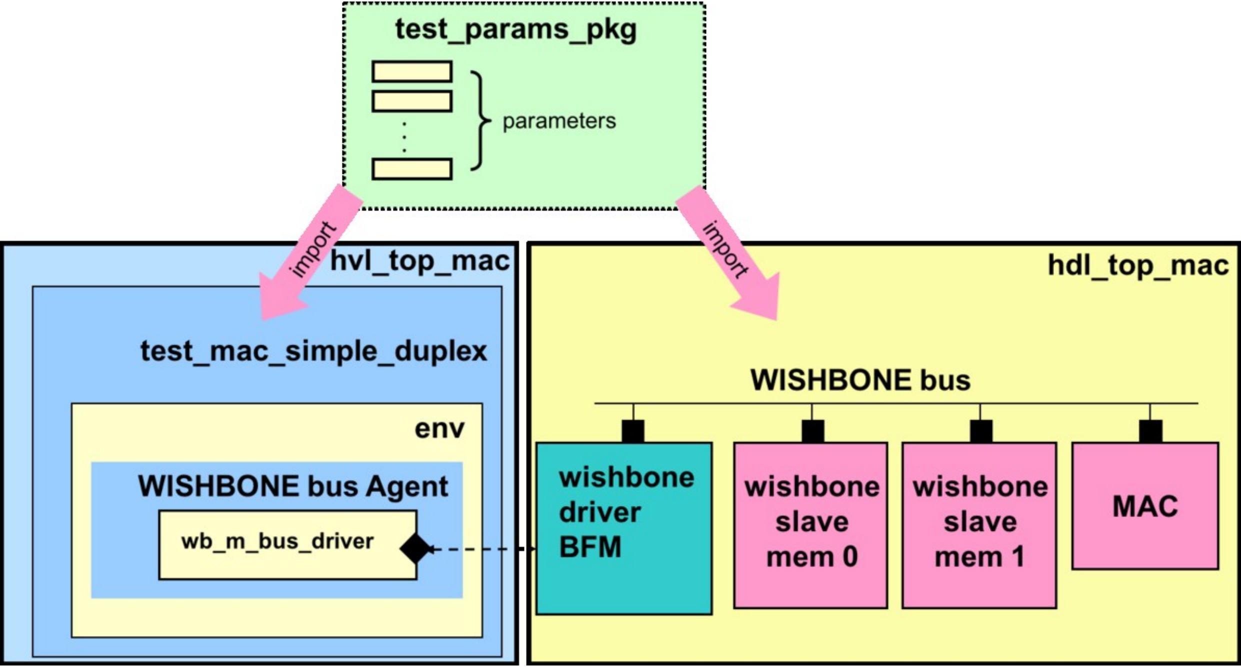
Parameter usage in the test class of the testbench to set the configuration object values for the WISHBONE bus slave memory is shown below. Note that instead of the numeric literal 32'h00100000, an expression involving the named DUT parameter mem_slave_size is used to assign the address value.

```
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

In case of multiple instances of parameter sets an instance-mnemonic based naming convention can be used for distinguishing instances, or alternatively a parameterized class-based approach to distinguish the parameter sets by parameter specialization.



Note that now two instances of the Wishbone slave mem exist. Each has its own parameter specialization. This is handled through a parameterized class which specifies the parameters and their default values. The actual parameter values are then set for each instance 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 use the parameters mem_slave_size and mem_slave_wb_id of the specialization WISHBONE_SLAVE_0 or WISHBONE_SLAVE_1 in the above code, use the following syntax: name_of_specialization::parameter_name as

illustrated below.

```
module hdl_top_mac;
  ...

  import test_params_pkg::*;

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

Further discussion on parameters and reuse is provided here.

Accessing Configuration Resources from a Sequence

Sequences often need access to testbench resources such as register models or configuration objects. This is best done using the uvm_config_db to retrieve a resource, as the first action of the body() method of a sequence base class that is extended by other sequences.

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 both creating a scope string, based on the m_sequencer location in the testbench hierarchy or the sequence "pseudo" location in the testbench hierarchy, respectively. The small difference between these two methods is as follows. With the first method, m_sequencer.get_full_name() is called when m_sequencer is passed as the argument to the get() call, yielding the path to the sequencer in the testbench hierarchy. An example of this might be "uvm_test_top.env.my_agent.sequencer". With the second method, get_full_name() is called on the sequence, not the sequencer. If the sequence was started on a sequencer (i.e. the

m_sequencer handle is not null), this get_full_name() call returns the path to the sequencer **with** the sequence name appended. An example of this might be "uvm_test_top.env.my_agent.sequencer.my_seq". This can be useful to target specific configuration information for a specific sequence running on a specific sequencer.

The third method relies on users agreeing to a naming convention for different configuration domains, which can work well within a particular project or workgroup but may cause reuse issues due to name clashes.

A full example of a base sequence implementation is shown below. A derived sequence must call the base sequence body() method to ensure that resource handles are set up before starting the stimulus process.

```
//
// Sequence that needs to access a testbench resource via the
configuration space
//
// Note that this is a base class; any class extending it must 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 env configuration object - 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

//
```

```
// A derived sequence:
//
class initialization_seq extends register_base_seq;
  `uvm_object_utils(initialization_seq)

  task body;
    super.body(); // assign the resource handles
    ... // Sequence body code
  endtask: body

endclass: initialization_seq
```

Another example of using this technique is to access a virtual interface within a configuration object to wait for hardware events.

Macro Cost-Benefit Analysis

Macros can be useful to reduce repetitive typing of small pattern-like code segments, to hide implementation differences or limitations among the simulators from different vendors, or to make critical code segments less error-prone for reuse. Many of the macros in UVM meet these criteria, but not all. While the benefits of macros in general may be obvious and immediate, often the costs associated with their usage is not transparent and may well become problematic at a later point when code changes are increasingly obtrusive.

The topic of macro usage is explored in ample detail in the DVCon 2011 paper entitled OVM-UVM Macros-Costs vs Benefits DVCon11 Appnote (PDF) [1]:

- It examines the hidden costs incurred by some macros, including code bloat, low performance, and debug difficulty.
- It identifies which macros provide a good cost-benefit trade-off, and which do not.
- It shows how to replace high-cost macros with simple SystemVerilog code.

The recommendations are summarized as follows:

Macro Type	Description
`uvm_*_utils	Always use. These register the object or component with the UVM factory. While not a lot of code, registration can be hard to debug if not done correctly.
`uvm_info** **`uvm_warning **`uvm_error** **`uvm_fatal**	Always use. These can significantly improve performance over their function counterparts (e.g. uvm_report_info).
`uvm_*_imp_decl	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.
`uvm_field_*	Do not use. These inject lots of complex code that substantially decreases performance and hinders debug.
`uvm_do_*	Do not use. These unnecessarily obscure a simple API and are best replaced by a user-defined task, which affords far more flexibility.
`uvm_sequence_utils** **`uvm_sequencer_utils **other related macros**	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.

References

[1] http://verificationacademy.com/go/resource/135

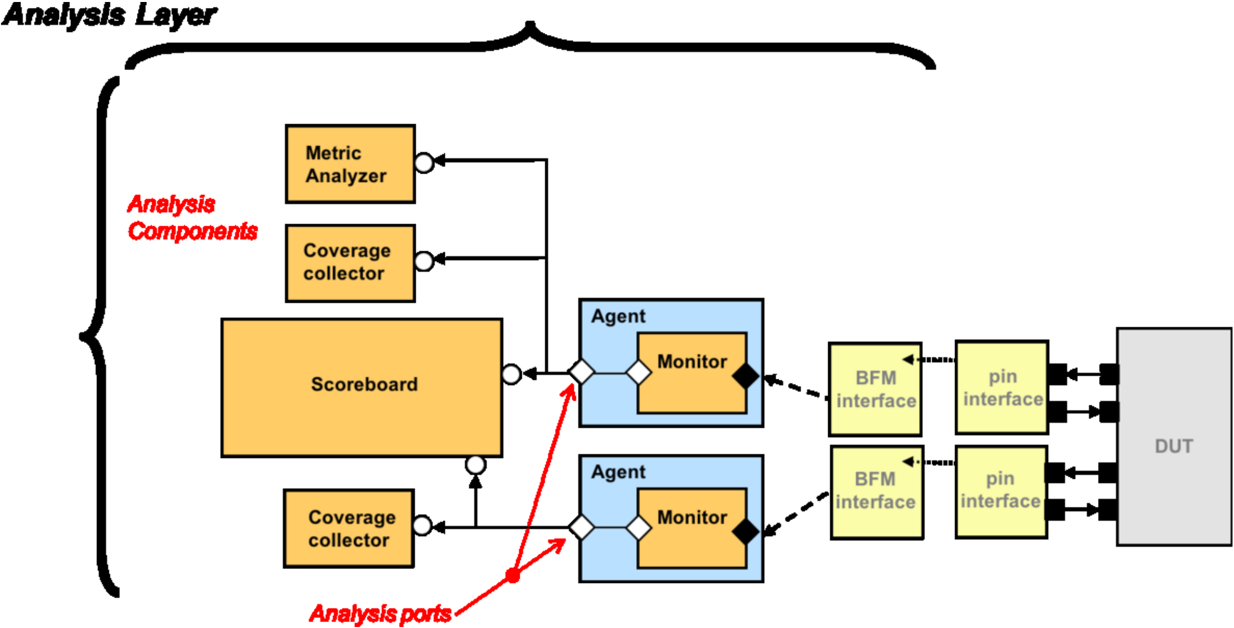
Analysis Components & Techniques

Analysis Components and Techniques

Topic Overview

Simply put, design verification involves two primary aspects: (i) the generation of input stimulus to put the design in a desirable, known state, and (ii) the analysis of corresponding results in the form of design outputs in response to the stimulus.

The analysis portion of a testbench is made up of components that observe behavior and make a judgement whether or not it is as expected in accordance with a design specification, thus providing confidence that the design conforms to its specification. Examples of specified behavior include functional behavior, performance, as well as power utilization.



Monitoring DUT Activity

The analysis part starts its decision process by observing the DUT responses. This is done by one or more monitors (proxy and BFM) that observe the signal-level activity on the DUT interfaces (via virtual interfaces). A UVM monitor converts signal-level activity into TLM transaction objects and broadcasts these to interested analysis components - i.e. subscribers - via analysis port connections. These subscribers capture the transactions and conduct their analysis.

Scoreboards

Scoreboards are analysis components that collect the transactions sent by a monitor and perform specific (often comparative) analysis computations to determine whether or not the design is functioning as expected. A good scoreboard architecture separates its tasks into two areas of concern: prediction and evaluation.

A predictor model, sometimes referred to as "Golden Reference Model", receives the same stimulus transactions as the DUT and computes known good response transactions. The scoreboard evaluates the predicted with respect to

the actual responses. The most common evaluation technique uses a comparator of some sort, capable of comparing transaction streams in in-order or a particular out-of-order fashion.

Coverage Collectors

Coverage information is paramount to answer the questions "Are we done (testing) yet?" or "Have we done enough (testing) yet?"

Coverage collectors are analysis port subscribers that sample observed transactions and activity into SystemVerilog functional coverage groups. The coverage data collected from each test is stored into a shared coverage database used to determine overall verification progress.

Metric Analyzers

Metric Analyzers watch and record non-functional behavior such as timing/performance and power utilization. They generally exhibit a standard architecture. Depending on the number of observed transaction streams, they are implemented as a uvm_subscriber or with analysis exports. Metric analyzers perform ongoing calculations during the run phase, and/or during the post-run phases.

Analysis Reporting

All data for analysis is collected during simulation. Most analysis takes place dynamically during the course of the simulation run (i.e. on-the-fly), though some analysis may be performed after the UVM run_phase ends when timed simulation completes. UVM provides three phases for this purpose: extract_phase, check_phase, and report_phase. These post-run phases provide components with the option to extract relevant data collected on-the-fly during the run, perform a post-run check, and finally produce comprehensive reports on all analysis results.

Analysis Port

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.

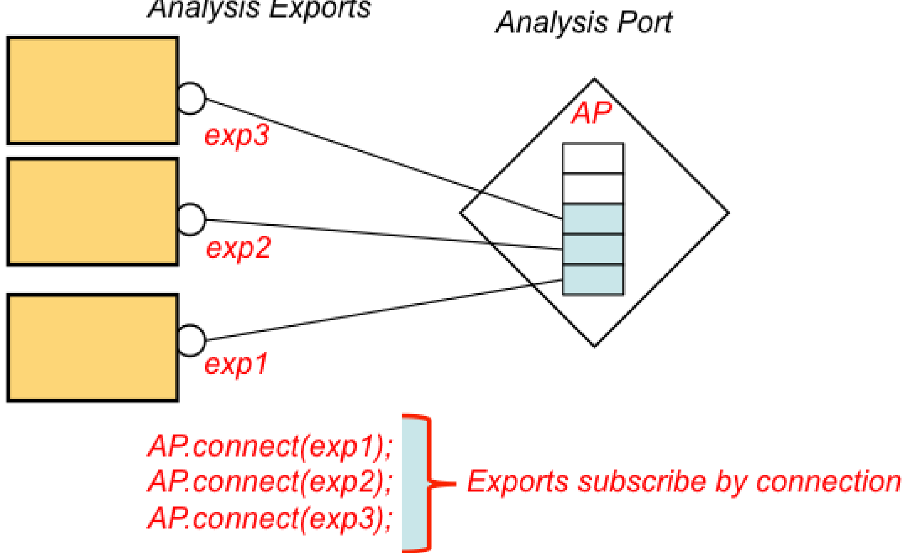
This connection topology implementation behavior lends itself to an OOP design pattern called the "observer pattern". In this pattern, interested "observers" register themselves with a single information source. There is no minimum or maximum number of observers (e.g. the number of observers could be 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 requirement of non-interference: analysis ports, analysis exports, and analysis fifos.

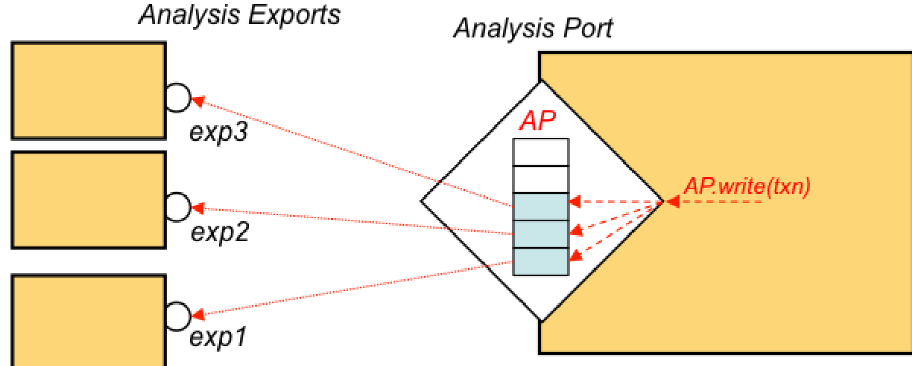
Detail

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.



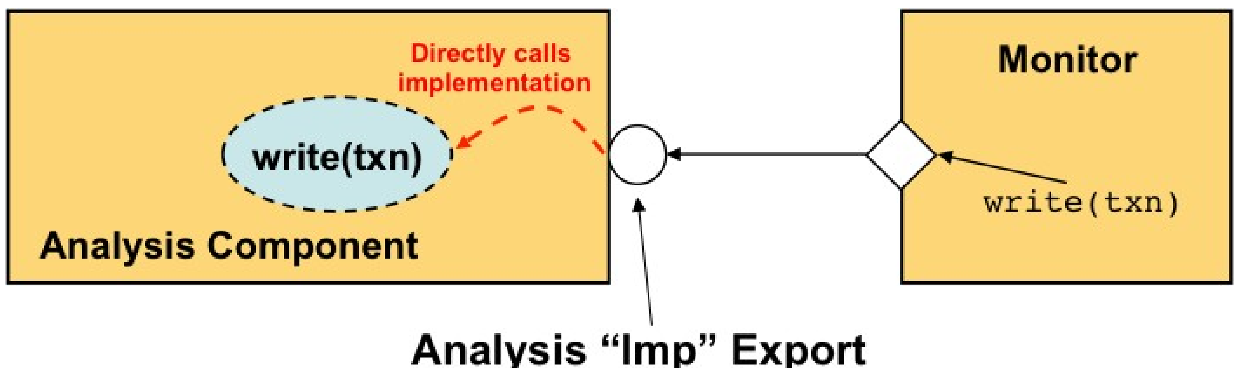
Analysis Port broadcasts to all subscribers

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.

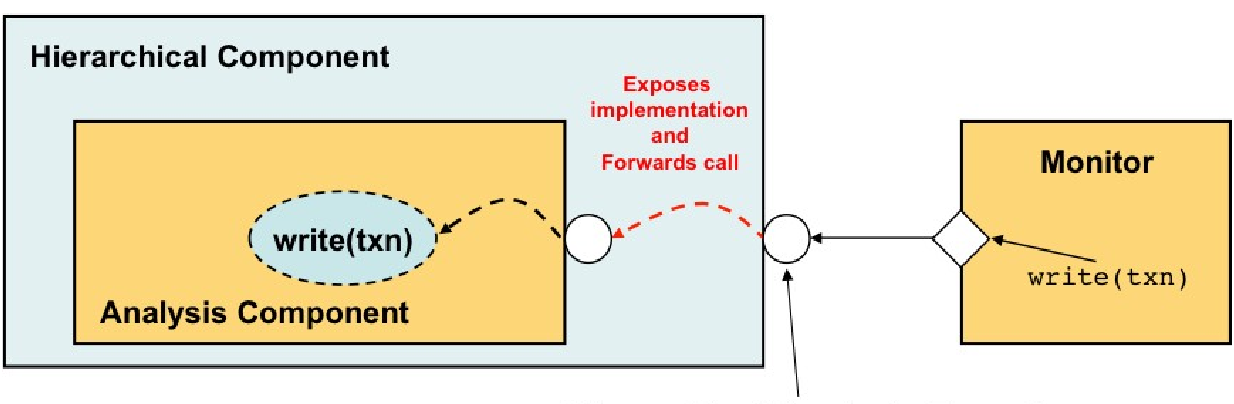
Analysis Exports

You register an analysis export as an observer with an analysis port by passing the export as the argument to the analysis 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.

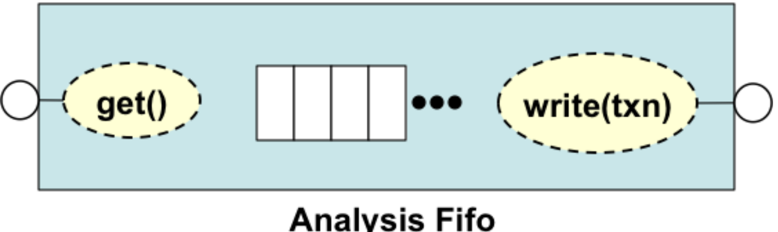


A hierarchical export is used for hierarchical connections, where the component that implements the write() function is a hierarchical child of the component with the export. A hierarchical export forwards the call to write() to the child component.

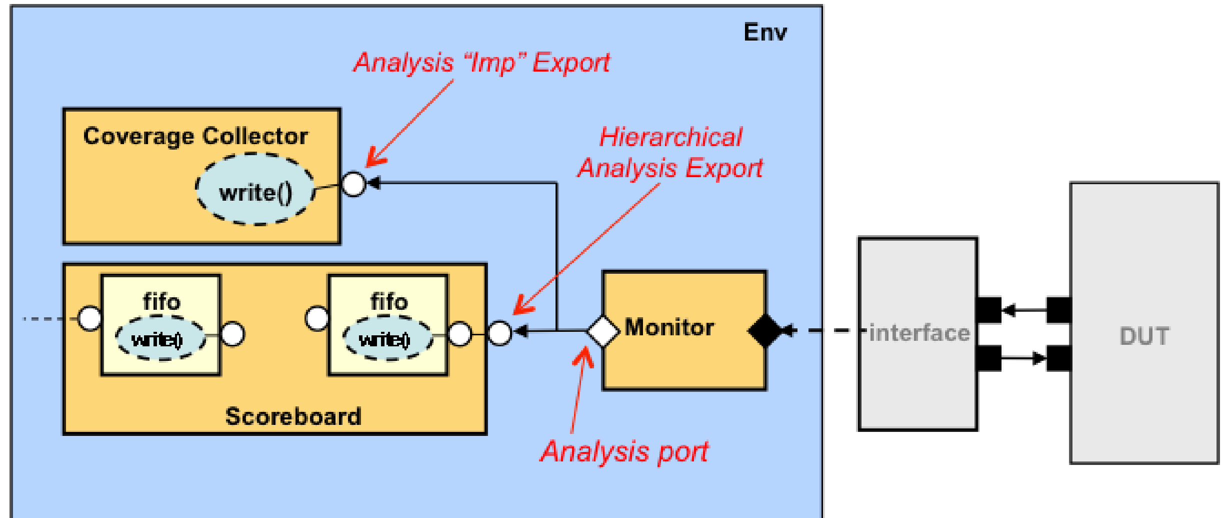


Analysis Fifos

UVM provides a pre-built component called an analysis fifo that has an "imp"-style analysis export and an implementation of the write() function 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, and so it also has all of the exports and associated 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 with a single input stream, one can extend the uvm_subscriber class. For components that have multiple input streams, one can either directly implement "write()" functions and provide "imp" exports, or one can expose write() function implementations of hierarchical children by providing hierarchical exports. The preferred choice u=is generally dependent on the intended functionality of a component as well as one's preferred coding style.

In the diagram above, the Coverage Collector extends the uvm_subscriber class which has an analysis "imp" export. The extension then implements the write() function.

The Scoreboard has two input streams of transactions and uses embedded Analysis fifos to buffer incoming transactions. In this case the write() method is implemented in the fifos, hence the Scoreboard has hierarchical analysis exports to expose the write() method externally.

Analysis Connections

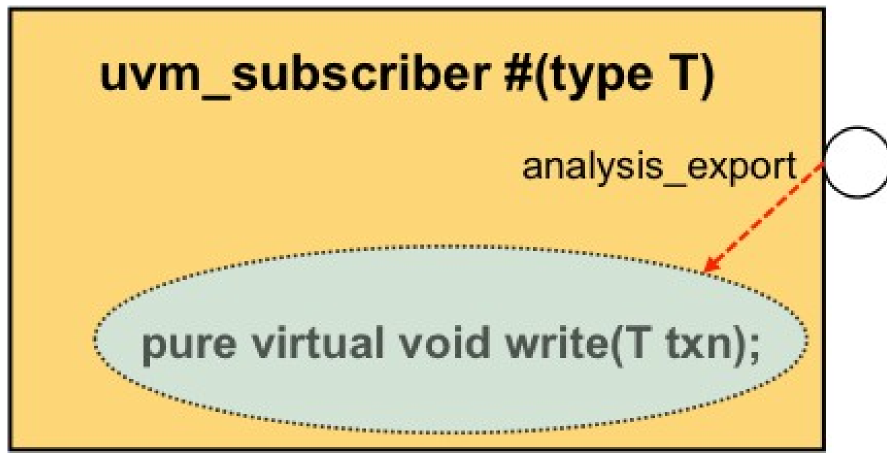
Overview

An analysis component such as a Monitor sends transactions to another analysis component through a TLM connection which is a chain of objects where each calls the write(t) function in the next. The monitor or other analysis component starts the chain by calling a write(t) function in the analysis port object, which calls write(t) in the next analysis object(s), until the final object calls write(t) in the component. The final TLM object is called an analysis imp as it calls the component that has the implementation of the write(t) function.

The chain of calls starts with an analysis port (uvm_analysis_port) which may be connected to another analysis_port to move up in the testbench hierarchy (Eg. monitor level to agent level). An analysis port may also be connected to an analysis export (uvm_analysis_export). This connection is made when at the same level of a testbench hierarchy (Eg. in the environment). When moving back down the hierarchy, the analysis export can be connected to another export or an analysis imp (uvm_analysis_imp). The imp ends the chain of calls.

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 must 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 = $get_coverage();
   `uvm_info("COVERAGE",$sformatf("%0d Packets sampled,  Coverage =
%f%% ",
                     pkt_cnt,current_coverage), UVM_MEDIUM)
 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 first approach using the `uvm_analysis_imp_decl macro is recommended as it provides better performance and uses less memory. The second approach using instances of uvm_tlm_analysis_fifo is a more traditional approach where a FIFO is created and testbench threads are spawned.

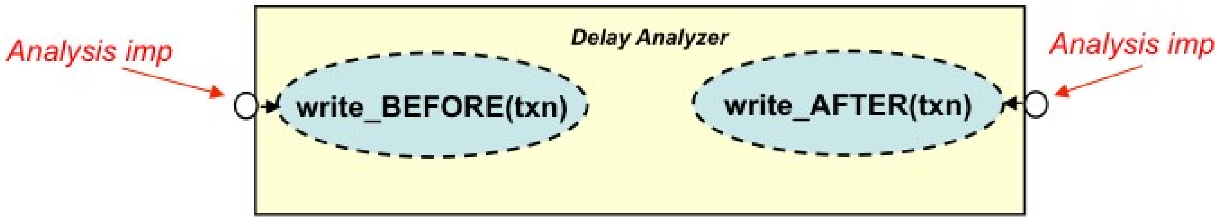
In one approach, you implement all the write() functions directly, and declare imp connections in the containing component. Because the write() functions must be non-blocking, you cannot directly perform any synchronization with other streams. If synchronization is required, you must deposit the transaction 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. The `uvm_analysis_imp_decl macro is creating an extension of the uvm_analysis_imp class with the _SUFFIX used in the extension name that defines the write_SUFFIX function.

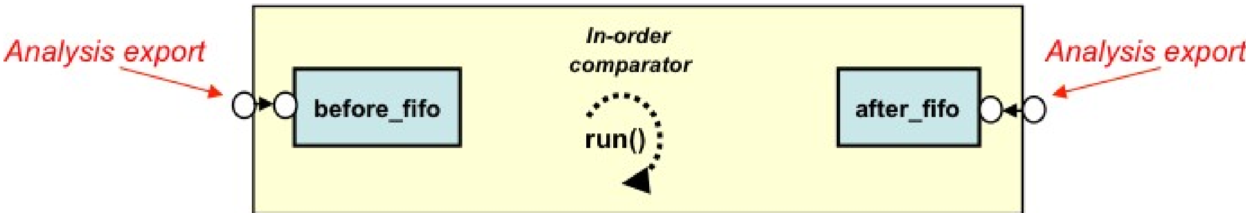



```
// Declare the suffixes that will be appended to the imps and functions
`uvm_analysis_imp_decl(_BEFORE)
`uvm_analysis_imp_decl(_AFTER)

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);
```

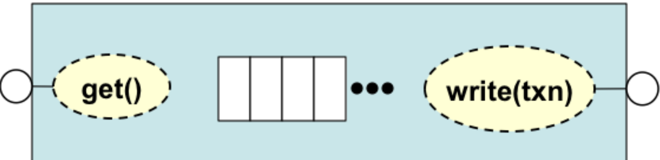
```
    real delay;
    delay = $realtime - last_a_time; last_a_time =
    $realtime; m_after.push_back(delay);
  endfunction

  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".
    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 );
    if (longest_a_delay > 100.0) begin
      `uvm_warning("Delay Analyzer", $sformatf("Transaction delay too long: %5.2f",
longest_a_delay))
    end
    if (longest_b_delay > 100.0) begin
      `uvm_warning("Delay Analyzer", $sformatf("Transaction delay too long: %5.2f",
longest_a_delay))
    end
  endfunction

  function void report_phase( uvm_phase phase );
   `uvm_info("Delay Analyzer", $sformatf("Longest BEFORE delay:
%5.2f", longest_b_delay), UVM_LOW)
   `uvm_info("Delay Analyzer", $sformatf("Longest AFTER delay:
%5.2f", longest_a_delay), UVM_LOW)
  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.



Analysis Fifo

```
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 );
```

```
   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
       `uvm_error("Comparator Mismatch",
                  $sformatf("%s does not match %s",
before_txn.convert2string(), after_txn.convert2string()))
       m_mismatches++;
     end else begin
       m_matches++;
     end
   end
 endtask

 function void report_phase( uvm_phase phase );
  `uvm_info("Inorder Comparator", $sformatf("Matches:    %0d",
m_matches), UVM_LOW)
  `uvm_info("Inorder Comparator", $sformatf("Mismatches: %0d",
m_mismatches), UVM_LOW)
 endfunction

endclass
```

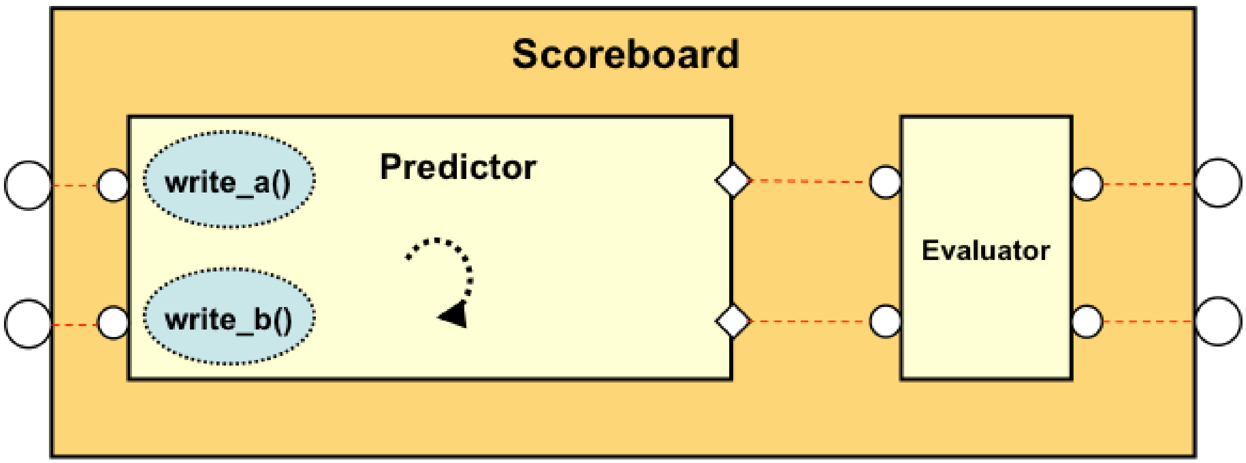
For more complicated synchronization needs, you would use a combination of multiple write_SUFFIX() functions which would place transaction data into some kind of shared data structure, along with code in the run() task to perform coordination and control.

Predictors

Overview

A Predictor is a verification component that represents a "golden" reference 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. When you send random stimulus into your DUT, you need an automatic way to check the result as you can no longer check the output by hand. A Predictor generates expected output that is compared to the actual DUT output to give a pass / fail.

Predictors in the Testbench Environment



Predictors are the part of the Scoreboard component that generates 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 );
```

```
  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
```

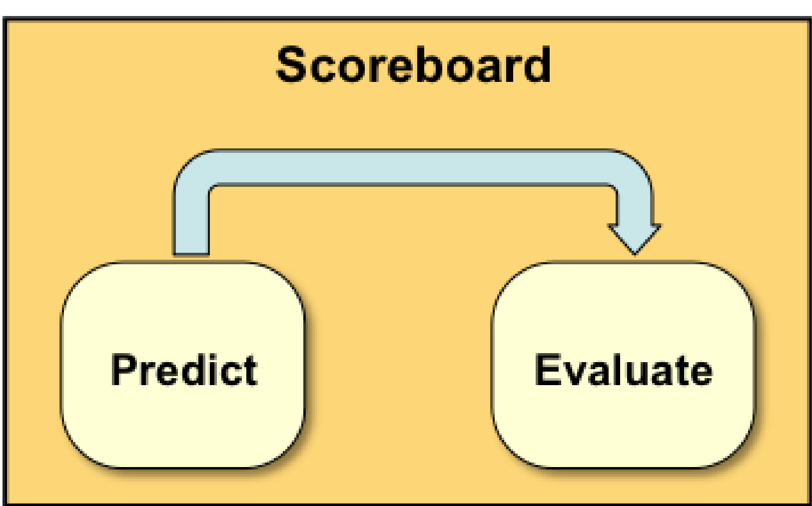
Predictor as a Proxy for the DUT

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.

Scoreboards

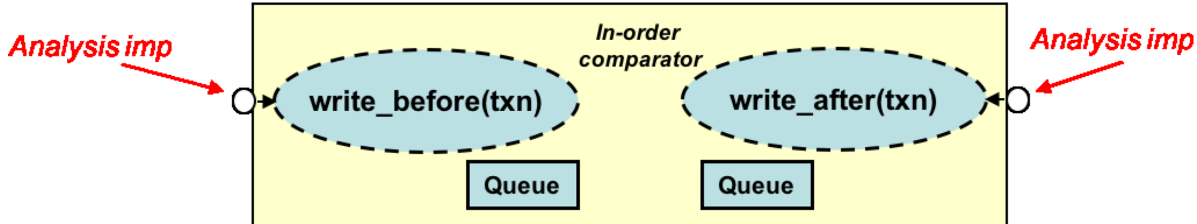
Overview

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

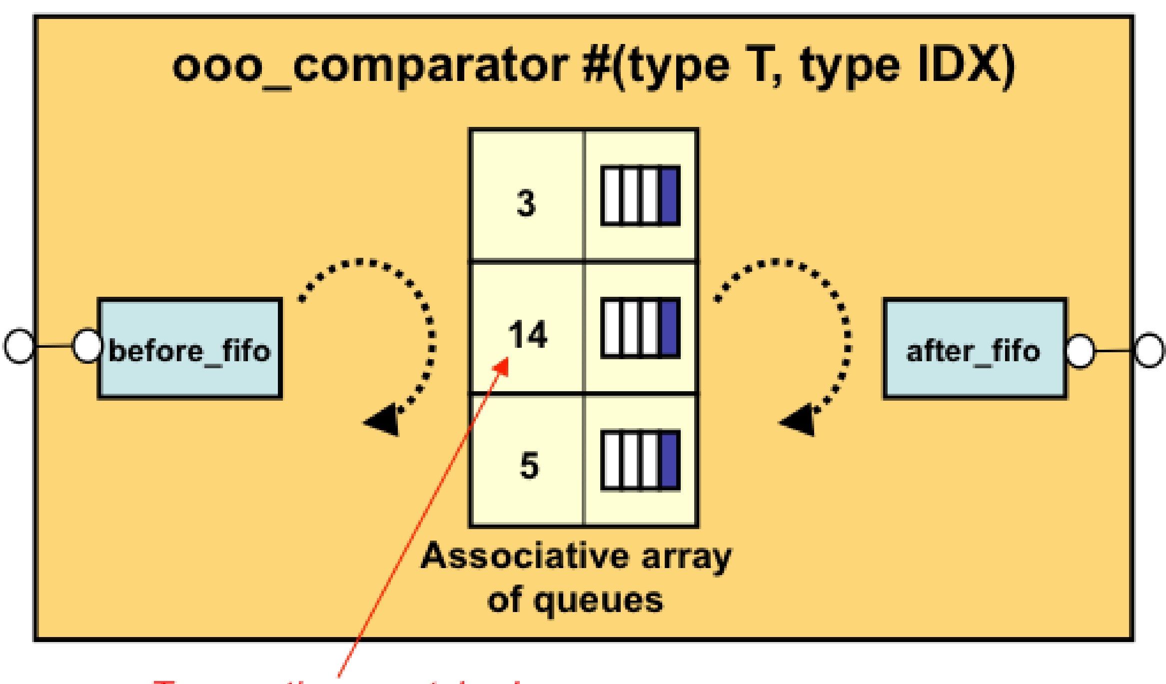


An in-order comparator assumes that matching transactions will appear in the same order from both expected and actual streams. It gets transactions from the expected and actual side and evaluates them. The transactions will arrive independently, but in order so transactions can be stored from the "before" side and then compared when a transaction arrives on the "after" side. 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_imp_before #(T, comparator_inorder) before_export;
  uvm_analysis_imp_after #(T, comparator_inorder) after_export;

  int m_matches, m_mismatches;
  protected T m_before[$];
  protected T m_after[$];

 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_export = new("before_export", this);
   after_export = new("after_export", this);
 endfunction

 protected virtual function void m_proc_data();
   T bef = m_before.pop_front();
   T aft = m_after.pop_front();
   if (!bef.compare(aft)) begin
     `uvm_error("Comparator Mismatch",
         $sformatf("%s does not match %s",
                bef.convert2string(),
                aft.convert2string()))
     m_mismatches++;
```

```
    end
    else begin
      m_matches++;
    end
  endfunction

  virtual function void write_before(T txn);
     m_before.push_back(txn);
     if (m_after.size())
       m_proc_data();
  endfunction

  virtual function void write_after(T txn);
     m_after.push_back(txn);
     if (m_before.size())
       m_proc_data();
  endfunction

  function void report_phase( uvm_phase phase );
   `uvm_info("Inorder Comparator", $sformatf("Matches:    %0d",
m_matches), UVM_LOW);
   `uvm_info("Inorder Comparator", $sformatf("Mismatches: %0d",
m_mismatches), UVM_LOW);
  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,
input bit is_before);
        T txn_data, txn_existing; IDX idx;
        string rs; q_of_T
        tmpq;
        bit need_to_compare;
        forever begin

      // 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

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.

Metric Analyzers

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_warning("Delay Analyzer",s);
   end
   if (longest_b_delay > 100.0) begin
     $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
     `uvm_warning("Delay Analyzer",s);
   end
 endfunction

 function void report_phase( uvm_phase phase );
  `uvm_info("Delay Analyzer", $sformatf("Longest BEFORE delay:
%5.2f", longest_b_delay), UVM_LOW);
  `uvm_info("Delay Analyzer", $sformatf("Longest AFTER delay:
%5.2f", longest_a_delay), UVM_LOW);
 endfunction

endclass
```

Post-Run Phases

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 exected 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

The Final phase is the very last UVM phase to be executed before the UVM executes $finish to bring the simulation to an end.

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_warning("Delay Analyzer",s);
   end
   if (longest_b_delay > 100.0) begin
      $sformat(s, "Transaction delay too long: %5.2f",longest_a_delay);
      `uvm_warning("Delay Analyzer",s);
   end
endfunction

function void report_phase( uvm_phase phase );
  `uvm_info("Delay Analyzer", $sformatf("Longest BEFORE delay:
%5.2f", longest_b_delay), UVM_LOW);
  `uvm_info("Delay Analyzer", $sformatf("Longest AFTER delay:
%5.2f", longest_a_delay), UVM_LOW);
```

```
  endfunction

  function void final_phase( uvm_phase phase );
   my_summarize_test_results();
  endfunction

endclass
```

End Of Test Mechanisms

End-of-Test and Objection Mechanisms

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
```

phase_ready_to_end() **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, unless set_propagate_mode(0) is used, in which case the propagation is directly to uvm_test_top.

- 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 test's phase methods to have the test control end-of-phase - usually when the execution of a sequence (or set of sequences) has completed.
- 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 than 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 hierachy, 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

UVM Sequences

Sequences in UVM

UVM Sequences provide an object-oriented mechanism to specify stimulus generation at the transaction level, which makes test writers more efficient and effective while also promoting abstraction and reuse. You will find an introduction to sequences in the Sequence Overview page.

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 eachother, 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.

Sequence Items

The UVM stimulus generation process is based on sequences controlling the behavior 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.

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.

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 behavior 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 optimizes 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

Introduction

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 summarized 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 a 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
to_struct	-	Optional: returns a struct representation of the object
from_struct	-	Optional: populates the data members of the object from the struct input

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 purpose is to print out a string representation of an uvm data object using one of the uvm_printer policy classes. The simplest way to implement the method is to set the printer string to the value returned by the convert2string() method.

```
function void do_print(uvm_printer printer);
  printer.m_string = convert2string();
endfunction: do_print
```

An alternative, higher performance version of this would use $display() to print the value returned by convert2string(), but this would not allow use of the various features of the various uvm_printer policy classes for formatting the data.

```
function void do_print(uvm_printer printer);
  $display(convert2string());
endfunction: do_print
```

To achieve full optimization, 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.

to_struct and from_struct

The to_struct() and from_struct() methods are optionally inserted into the transaction to convert the data members of the object into a packed struct representation which would be suitable for synthesis into an emulator. The struct definition itself is contained with a separate package which is shared between the simulation and the emulation worlds. This package with the struct definition is imported into the agent's package where the transaction class is `included. The shared package is also imported in any BFM where the struct is used.

The to_struct() and from_struct() functions are not defined in the UVM base class library. That is why they don't follow the do_*() convention that the other transaction functions follow.

The to_struct() function is called on the transaction object and returns a packed struct representation of the object. That packed struct can then be used directly to send information to an emulator.

```
function item_s to_struct();
  to_struct.addr     = addr;
  to_struct.data     = data;
```

```
  to_struct.injerr  = injerr;
endfunction : to_struct
```

When information is received back from an emulator, it should be converted back to an object which can easily be passed around a UVM testbench. The from_struct() method performs that duty by populating the object with data from the packed struct.

```
function void from_struct(item_s item);
  addr    = item.addr;
  data    = item.data;
  injerr  = item.injerr;
endfunction : from_struct
```

Sequence API

Sequence API Fundamentals

A uvm_sequence is derived from an uvm_sequence_item and it is parameterized 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 initialized 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 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 sequencers 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 clearhow 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 initialization 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() means that they may or may not be called, depending on the arguments. It is therefore easier for the user if the pre_/post_body methods are not used.

Do not raise/drop objections from a sequence

It is much easier to coordinate the raising and dropping of objections from the test (see UVM Coding Guidelines), so objections should never be raised or lowered from a sequence. This rule extends to not using the set_automatic_phase_objection() method added in UVM1.2, since this causes the sequence to raise and lower an objection.

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

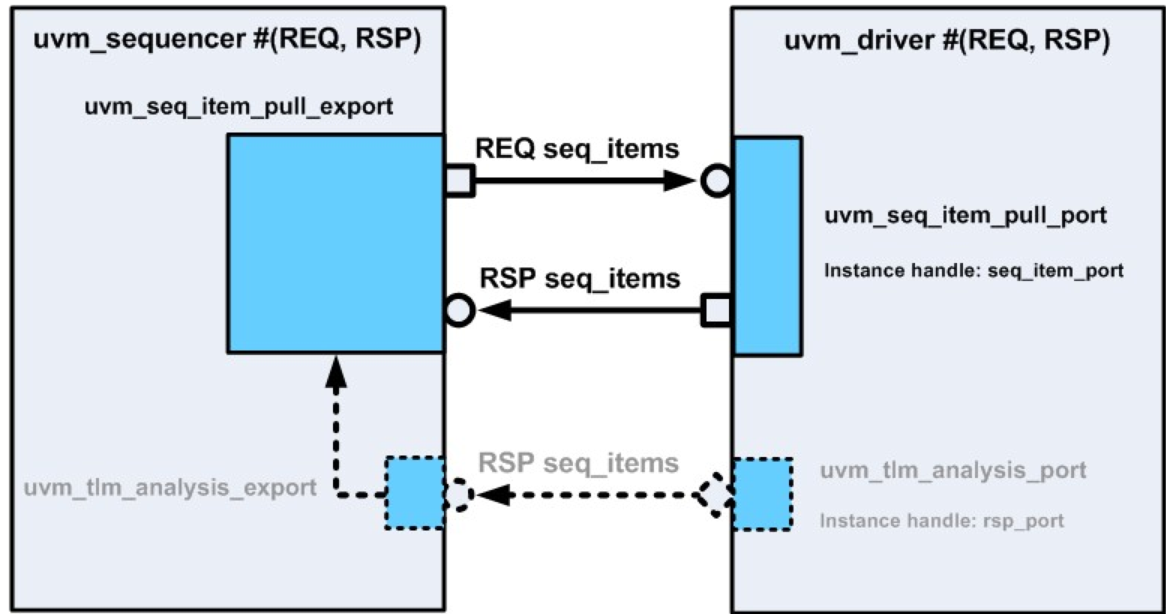
Sequence code should not explicitly consume time by using delay statements. They should only consume time by virtue of the process of sending sequence_items to a driver.

Justification:

Keeping this separation allows sequences to be reused as stimulus when an UVM testbench is linked to an emulator for hardware acceleration.

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 parameterized 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.



Sequencer-Driver Connections

The connection between the driver port and the sequencer export is made using a TLM connect method during the connect phase:

```
// Driver parameterized with the same sequence_item for request &
response
// response defaults to request
class adpcm_driver extends uvm_driver #(adpcm_seq_item);
....
endclass: adpcm_driver

// Agent containing a driver and a sequencer - uninteresting bits left
out
class adpcm_agent extends uvm_agent;
```

```
adpcm_driver m_driver;
adpcm_agent_config m_cfg;
// uvm_sequencer parameterized with the adpcm_seq_item for request &
response
uvm_sequencer #(adpcm_seq_item) m_sequencer;

// 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
```

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;
uvm_sequencer #(adpcm_seq_item) 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 parameterized class and it is parameterized with the type of the request sequence_item and the type of the response sequence_item. In turn, these parameters are used to parameterize the uvm_seq_item_pull_port. The fact that the response sequence_item can be parameterized 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 handshake 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.

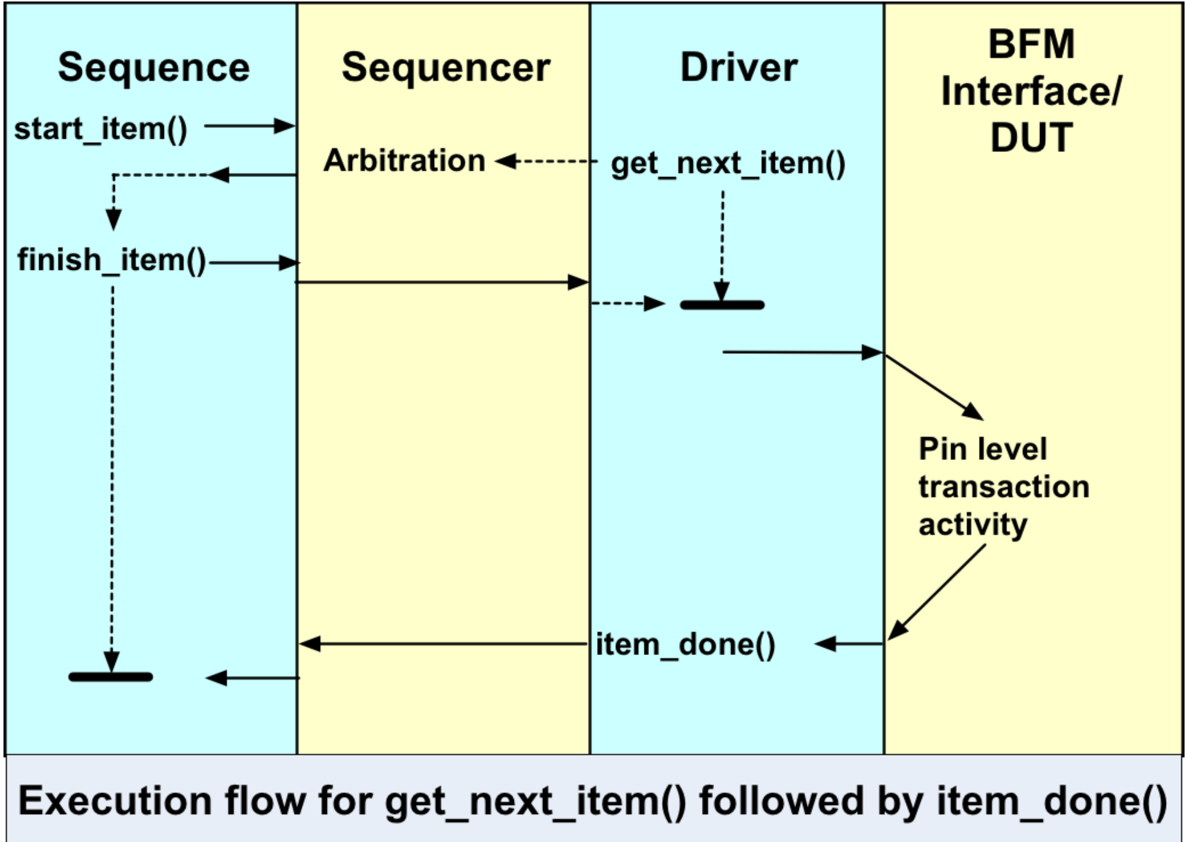
Note: The get_next_item(), get() and peek() methods initiate the sequencer arbitration process, which results in a sequence_item being returned from the active sequence which has selected. This means that the driver is effectively pulling sequence_items from the active sequences as it needs them.

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.



Execution flow for get_next_item() followed by item_done()

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

    //BFM handles all pin wiggling and population of req_item with
response data
    m_bfm.drive(req_item);

    seq_item_port.item_done(); // Signal to the sequence that the
driver has finished with the item
  end
endtask: run
```

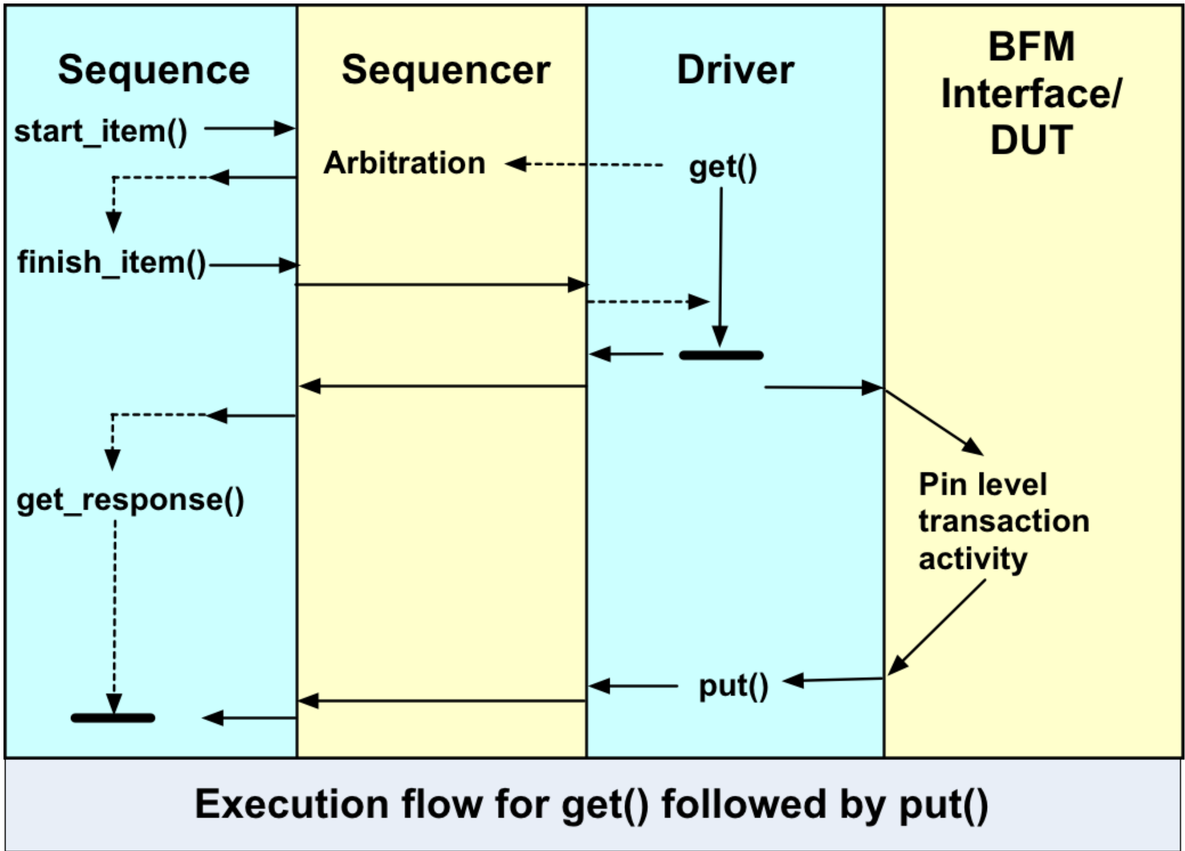
The corresponding sequence implementation would be a start_item() followed by a finish_item(). Since the driver proxy, driver BFM and the sequence are pointing to the same sequence_item, any data returning from the driver BFM can be referenced within the sequence via the sequence_item handle. In other 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_info("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.



Execution flow for get() followed by put()

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

  bit [15:0] rdata;
  bit error;

  forever begin
    seq_item_port.get(req_item); // finish_item in sequence is
unblocked

    //BFM handles all pin wiggling and returns response data as
separate arguments
```

```
    m_bfm.drive(req_item, rdata, error);

    $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 = rdata;
    end
    rsp_item.resp = 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 occuring 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_item.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
```

Generating Stimulus with UVM Sequences

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.

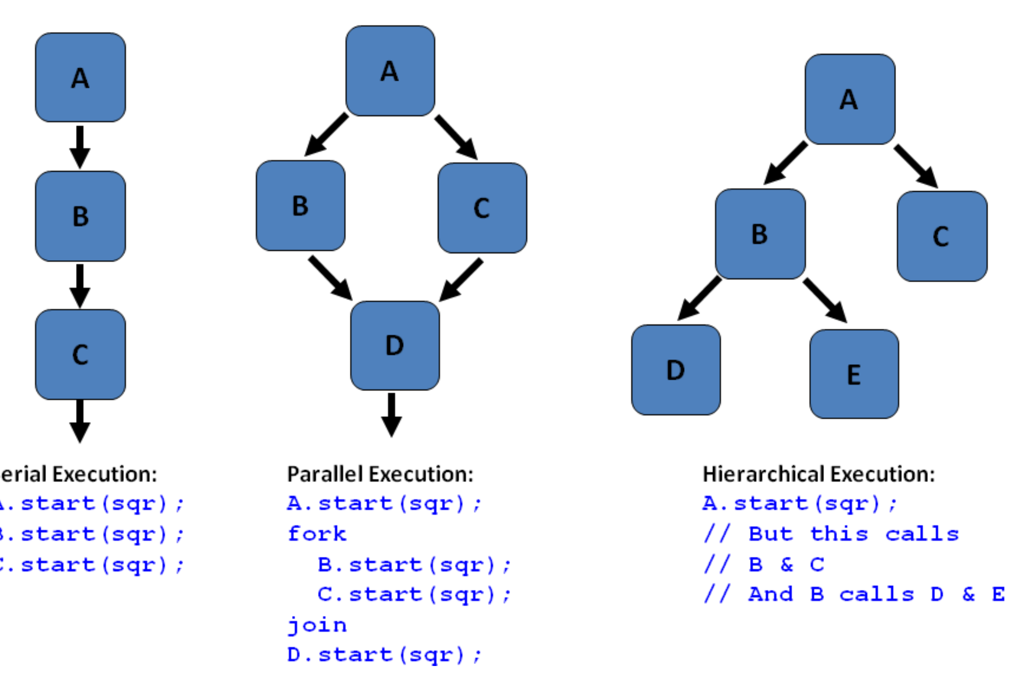
The use of sequences in the UVM enables a very flexible approach to stimulus generation. Sequences are used to control the generation and flow of sequence_items into drivers, but they can also create and execute other sequences, either on the same driver or on a different one. Sequences can also mix 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.

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
```

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 = "rand_order_seq");
  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.

Overriding Sequences and Sequence Items

Sometimes, during stimulus generation, it is useful to change the behavior 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:

An 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);
```

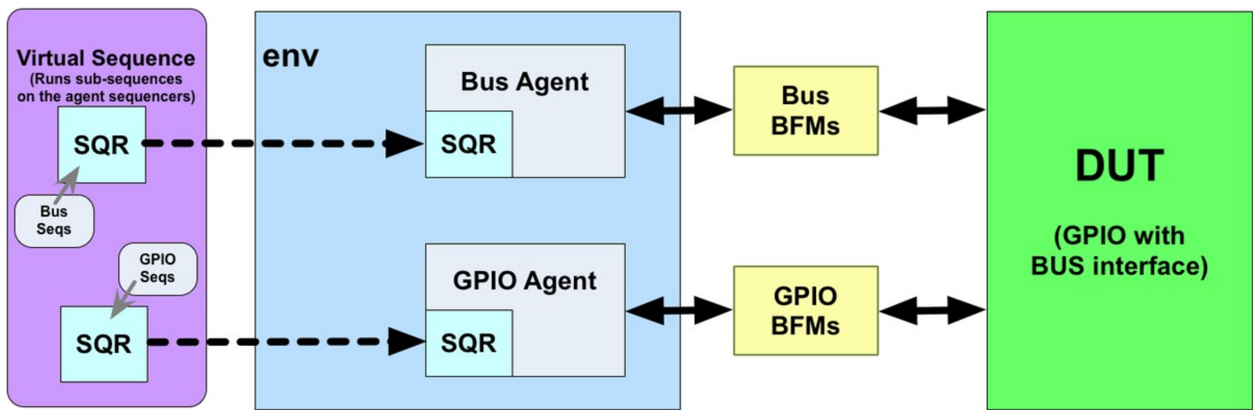
```
 // Set type override
 b_seq::type_id::set_type_override(c_seq::get_type());
 // Set instance override - Note the "path" argument see the line for
s_a creation
 // in the run method
 a_seq::type_id::set_inst_override(c_seq::get_type(), "bob.s_a");
endfunction: build

//
// 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
```

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.

Virtual Sequences

A virtual sequence is a sequence which controls stimulus generation using several sequencers. Since sequences, sequencers and drivers (proxy and BFM) are focused on point interfaces, almost all testbenches require a virtual sequence to coordinate 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 'coordinator 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");
```

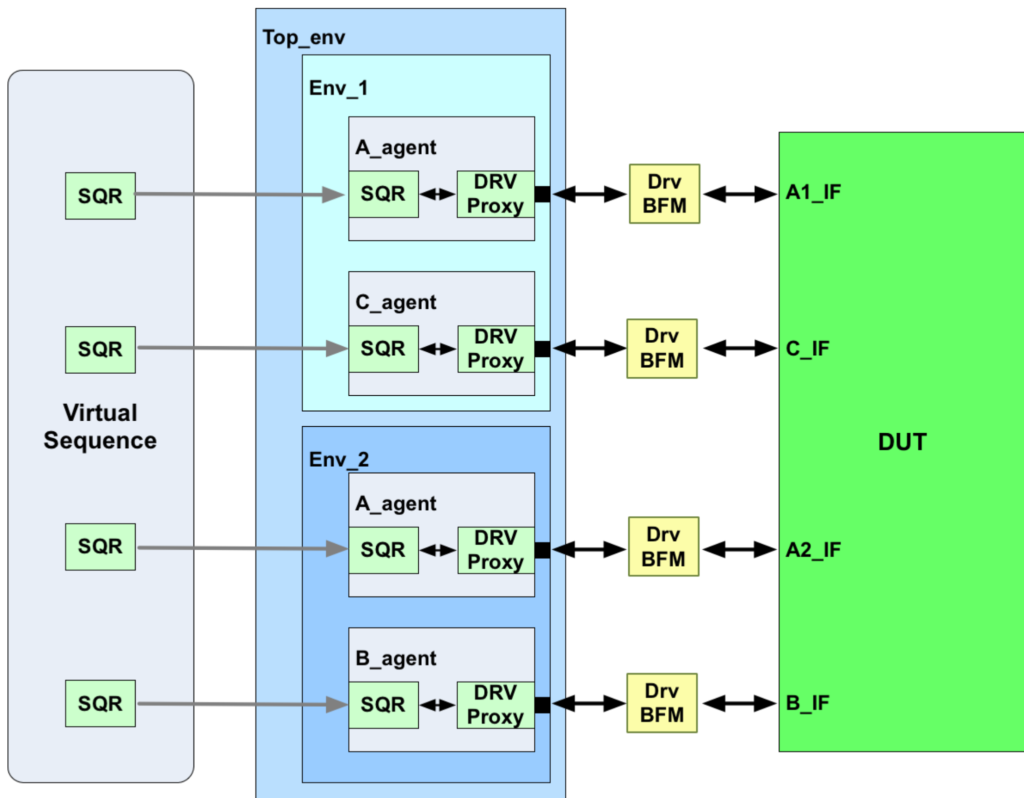
```
vseq.a_sequencer = env.subenv1.bus_agent.sequencer;
vseq.b_sequencer = env.subenv2.subsubenv1.bus_agent3.sequencer;

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 Initialization Methodology

In order to use the (U)OVM effectively, many organizations separate the implementation of the testbench from the implementation of the test cases. This is either a conceptual separation or a organizational 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.



Virtual Sequence Example

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 initialize 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 initialization 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 initialization 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.

Alternative Methods For Initializing A Virtual Sequence

In the preceding example, the virtual sequence is initialized 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 initialization 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 initialization method in a mapping package which imports the env package(s) and the virtual sequence package. This separates out the initialization from the test. Several initialization 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 initialization 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")
```

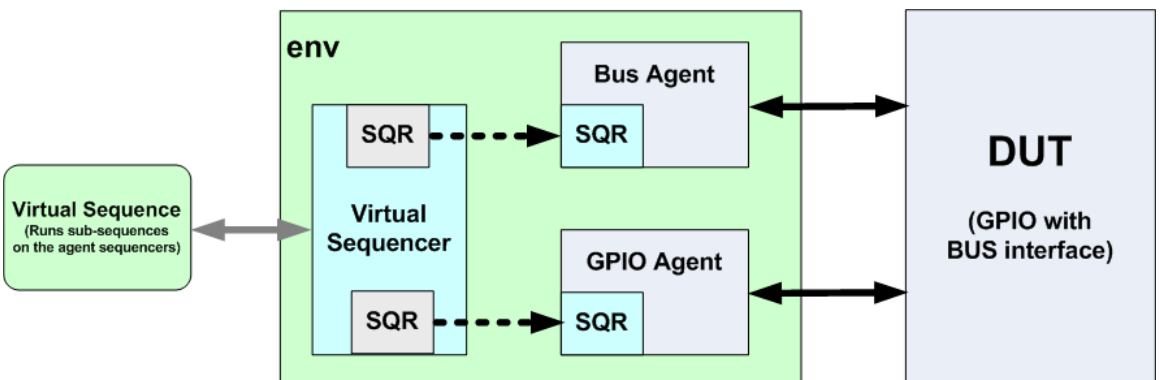
```
  else
    $cast(A, tmp[0]);
  //find the B sequencer in the testbench
  tmp.delete(); //Make sure the queue is empty
  uvm_top.find_all("*m_gpio_agent_h.m_sequencer_h", tmp);
  if (tmp.size() == 0)
    `uvm_fatal(report_id, "Failed to find mem sequencer")
  else if (tmp.size() > 1)
    `uvm_fatal(report_id, "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.

Virtual Sequencers (Not Recommended)

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 coordinate the stimulus across different interfaces and the interactions between them.



The Virtual Sequencer and the Virtual Sequence

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 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".

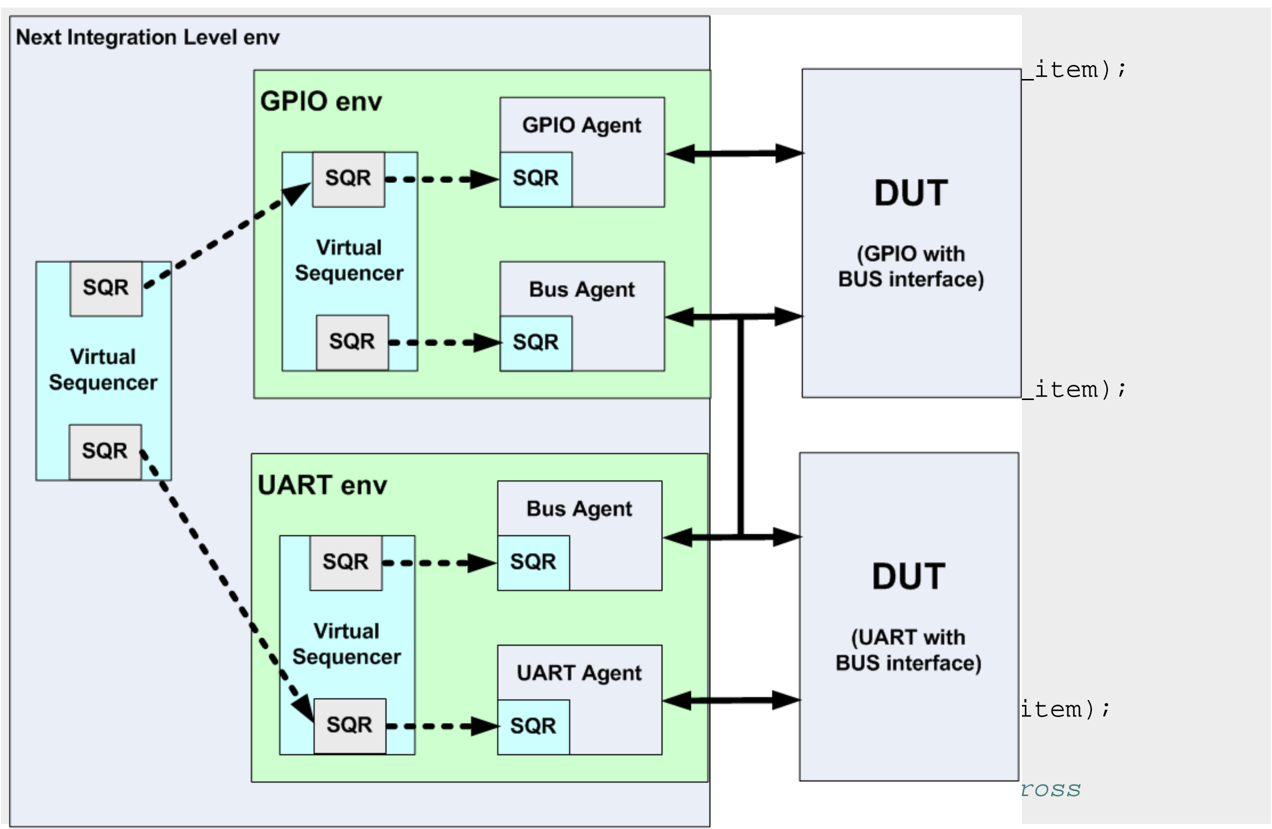
```
// Virtual sequencer class:
class virtual_sequencer extends uvm_sequencer #(uvm_sequence_item);

`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 at next level of integration (only one bus agent is active)

```
// 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

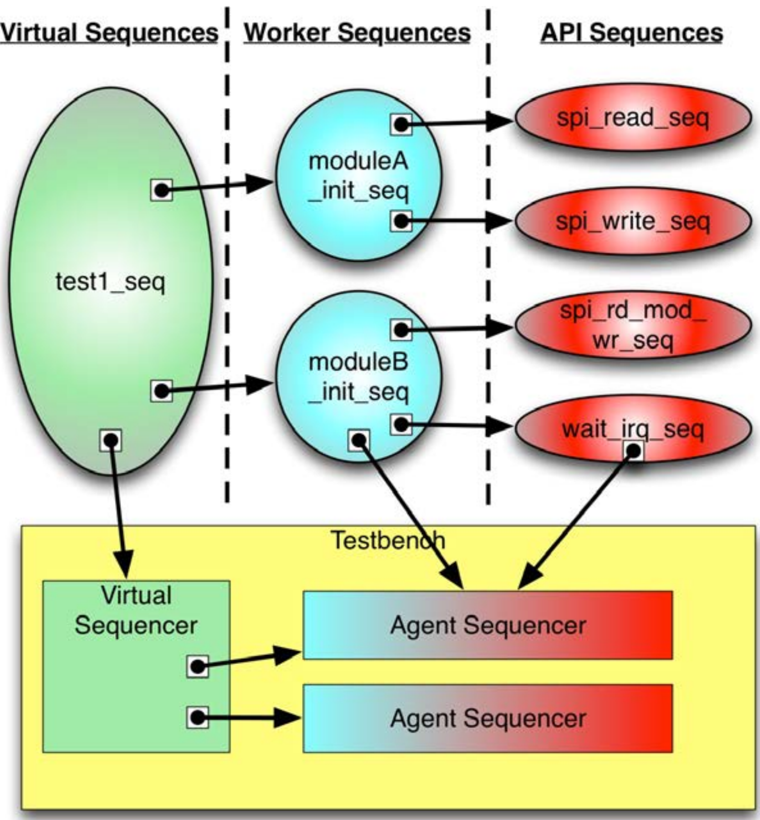
Virtual sequence implementations are based on the assumption that they can run sequences on sub-sequencers within agents. However, agents may be passive or active depending on the configuration of the testbench. In order to prevent test cases crashing with null handle errors, virtual sequences should 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_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
```

Hierarchical Sequences

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 coordinates 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;
    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

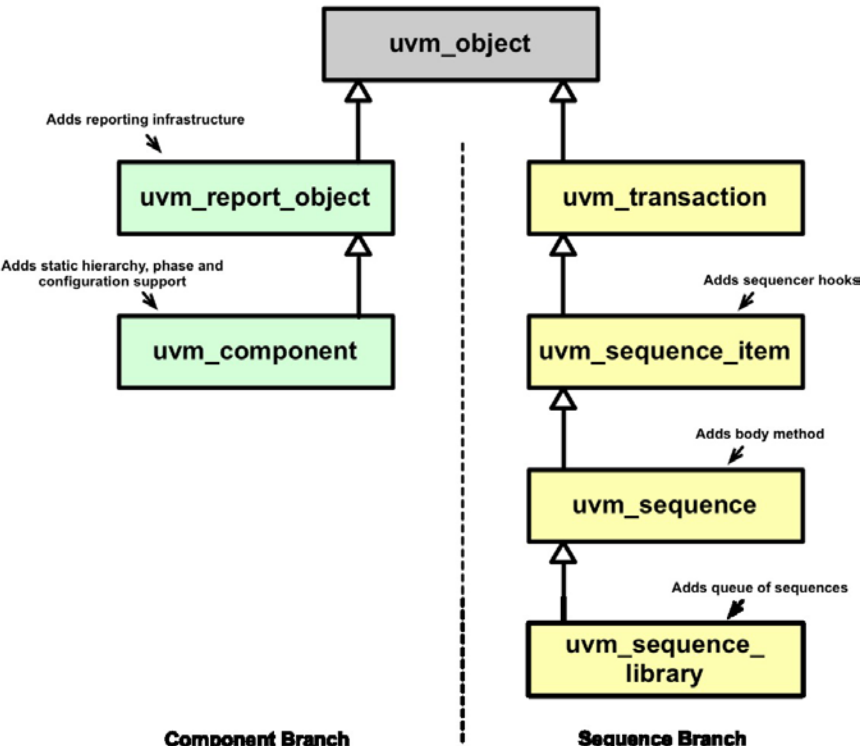
would look like this:

```
//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) 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
```

The Sequence Library

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.

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. 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);
   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 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()});

   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.

Sequences can be removed from the library using the remove_sequence() API.

```
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() );

    //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.set_min_random_count(5);
    seq_lib.set_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.

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.set_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
```

Sequence-Driver 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

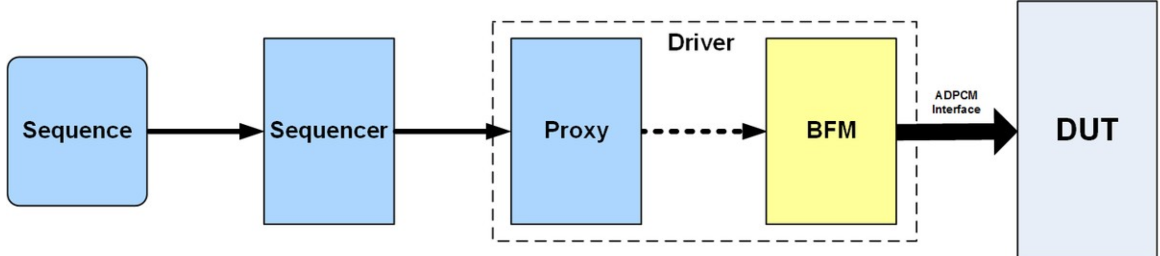
In applications such as fabrics and switches, requests are acknowledged but not responded to in order. The reason being that an accessed resource may already be busy or the path to it may already be in use, and the requesting master has its command accepted leaving it free to issue a new request. This 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.

Unidirectional Protocols

For a driver, composed of a BFM-proxy pair in the dual domain testbench, the driver proxy controls the flow of sequence_items by using get_next_item() to obtain the next sequence_item to be processed, and making the item_done() call only once 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 proxy.

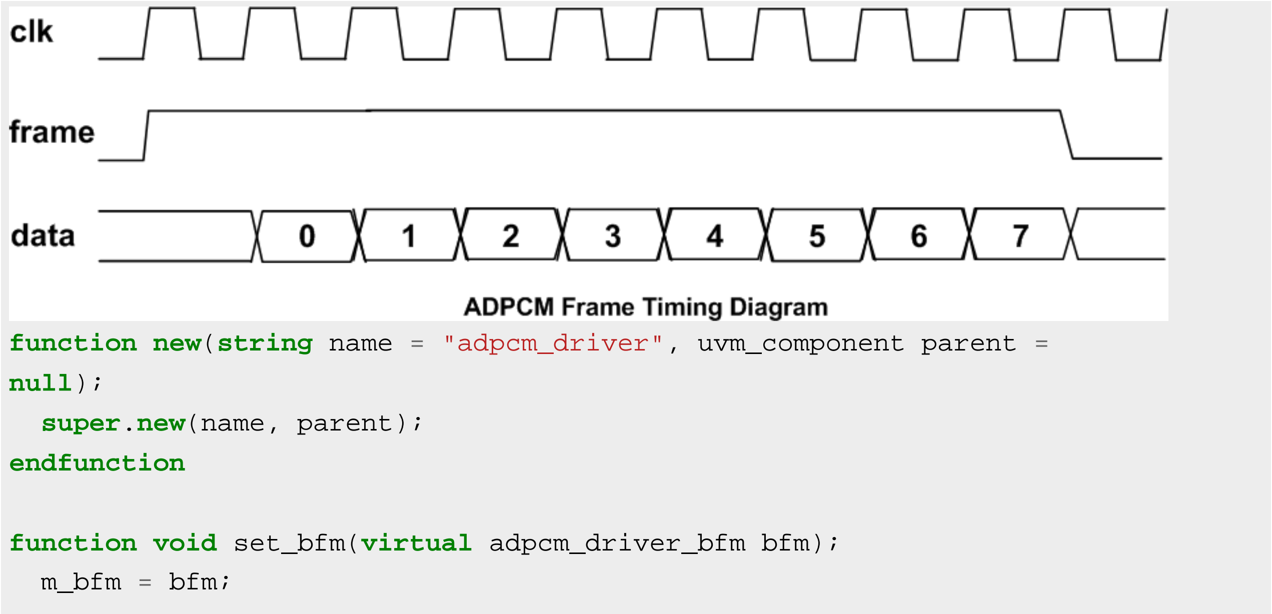


Unidirectional Data Flow Use Model Example - ADPCM

In the unidirectional non-pipelined sequence-driver use model, it is the data flow that is unidirectional. The sequence provides a series of request sequence_items to be driven on 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 BFM may also implement a handshake 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.



ADPCM Frame Timing Diagram

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

function void set_bfm(virtual adpcm_driver_bfm bfm);
  m_bfm = bfm;
```

```
endfunction: set_bfm

task run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(req); // Gets the sequence_item from
the sequence
        m_bfm.drive(req);
        seq_item_port.item_done(); // Indicates that the sequence_item has
been consumed
    end
endtask: run_phase

endclass: adpcm_driver
```

```
interface adpcm_driver_bfm (adpcm_if ADPCM);

import adpcm_pkg::*;

initial begin
    ADPCM.frame <= 0;
    ADPCM.data <= 0;
end

task drive(adpcm_seq_item req);
    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
endtask: drive

endinterface: adpcm_driver_bfm
```

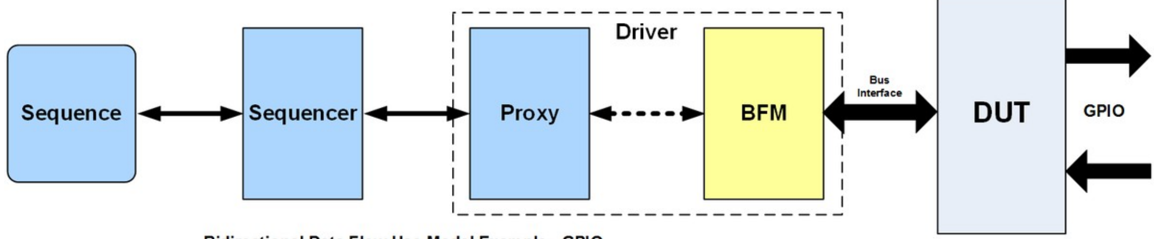
The sequence implementation in this case is a loop which generates a series of sequence_items. A variation on this theme would be for the sequence to actively shape the traffic sent rather than send purely random stimulus.

```
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");

  for (int i = 0; i < no_reqs; i++) begin
    start_item(req);
    if (!req.randomize()) begin
      `uvm_fatal("body", "req randomization failure")
    end
    finish_item(req);
    `uvm_info("ADPCM_TX_SEQ_BODY", $sformatf("Transmitted frame %0d",
i), UVM_LOW)
  end
endtask: body

endclass: adpcm_tx_seq
```

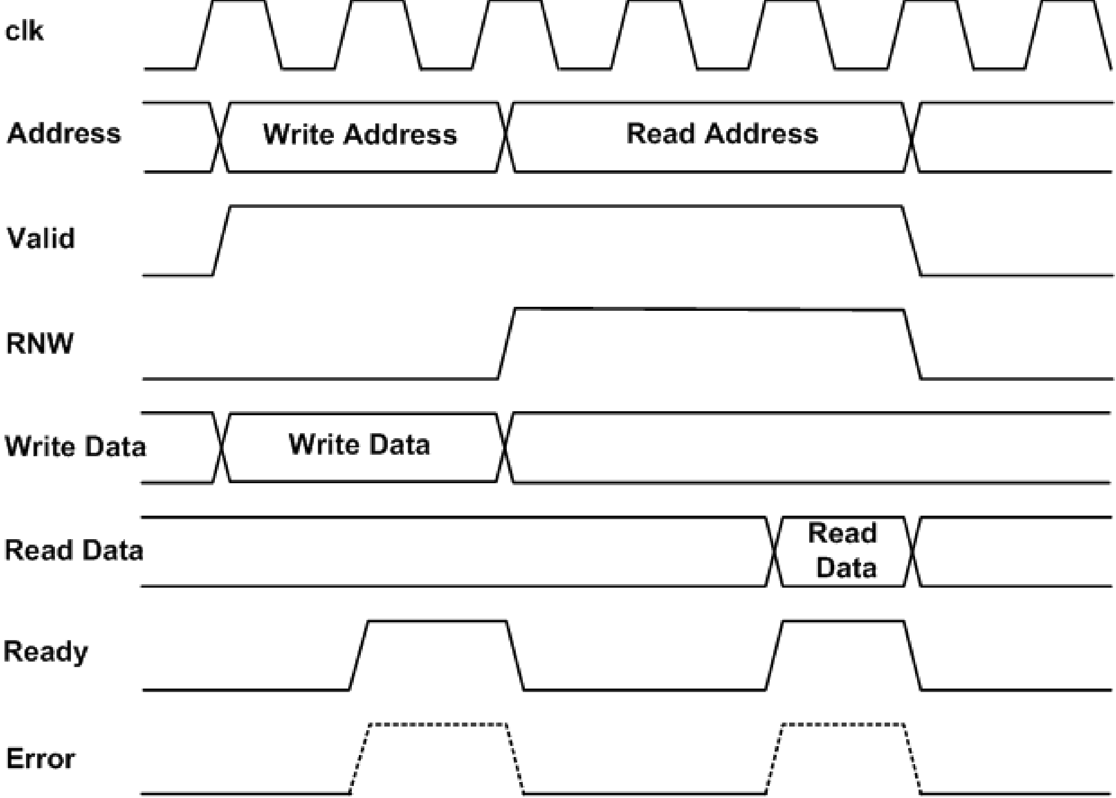
Bidirectional Protocols

For a driver, composed of a BFM-proxy pair in the dual domain testbench, one of the most common sequence-driver use cases is where the sequencer sends request sequence_items to the driver proxy, which then executes the request phase of the pin-level protocol through the driver BFM. The driver BFM in turn captures the response phase of the pin-level transaction, and returns the response back to the sequence via the driver proxy. In this scenario 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.



Bidirectional Data Flow Use Model Example - GPIO

To illustrate how this use model would be implemented, a DUT containing a GPIO and a bus interface is used. The bus protocol is shown in the timing diagram. The request phase of the transaction is initiated by asserting the valid signal, with the address and direction signal (RNW) indicating the type of bus transfer to take place. The response phase of the transaction is completed when the ready signal becomes active.



Coupled Request – Response Bus Protocol

The driver proxy that manages this protocol will collect a request sequence_item from the sequencer and then drive the bus request phase. The driver BFM waits until the interface ready line becomes active before capturing the response information, which consists here of the error status bit and the read data if a read has just taken place. The recommended way of implementing the driver proxy 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;

local virtual bidirect_bus_driver_bfm m_bfm;

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

function void set_bfm(virtual bidirect_bus_driver_bfm bfm); m_bfm = bfm;
endfunction: set_bfm

task run_phase(uvm_phase phase);
    m_bfm.wait_for_reset();

    forever begin
        seq_item_port.get_next_item(req); // Start processing req item
        m_bfm.drive(req);
        seq_item_port.item_done(); // End of req item
    end
endtask: run_phase

endclass: bidirect_bus_driver

interface bidirect_bus_driver_bfm (
      input     logic           clk,
      input     logic           resetn, output
      logic [31:0] addr, output logic [31:0]
      write_data, output logic  rnw,
      output logic              valid,
      input     logic           ready, input
                logic [31:0] read_data, input
                logic           error
);

import bidirect_bus_pkg::*; initial

begin
    BUS.valid <= 0;
    BUS.rnw <= 1;
end
```

```
task wait_for_reset();
  @(posedge BUS.resetn);
endtask: wait_for_reset

task drive(bus_seq_item req);
  repeat (req.delay) begin // Delay between bus transactions
    @(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; // Copy read data response
  end
  req.error = BUS.error; // Copy bus error response status

  BUS.valid <= 0; // End the pin level bus transaction
endtask: drive

endinterface: bidirect_bus_driver_bfm
```

Note that the driver proxy is sending back the response to the sequence implicitly through the pertinent response fields within the req sequence_item, by way of the driver BFM updating these fields as part of the drive() call, before the item_done() call. At the sequence end of the transaction, the sequence is blocked in the finish_item() call until the item_done() call occurs. Once the sequence is unblocked, its req handle is still pointing to the req object which has had its response fields updated by the driver BFM. This means that the sequence can reference the response fields 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_info("seq_body", req.convert2string(), UVM_MEDIUM);
  end
endtask: body

endclass: bus_seq
```

Alternative Implementation Option

Although an alternative implementation is discussed below, the recommended way to implement this sequence driver use model is as in the preceding code.

Driver put, Sequence get_response

The code in the driver proxy could use a get() method call to collect the request sequence_item, which would unblock the finish_item() call in the sequence execution. However, the driver proxy must then use the put() method to signal back to the sequence that the bus transfer has completed, and the sequence must use the blocking get_response() method call to wait for the driver proxy to notify completion of the transfer. Any response information from the pin-level bus transaction can be sent from the driver proxy to the sequence via the argument to the put() method.

```
// Alternative version of the driver proxy run method
task run_phase(uvm_phase phase);
  m_bfm.wait_for_reset();

  forever begin
    seq_item_port.get(req); // Start processing req item
    m_bfm.drive(req, rsp);
    rsp.set_id_info(req); // Set the rsp id = req id
    seq_item_port.put(rsp); // put returns the response
  end
endtask: run_phase
```

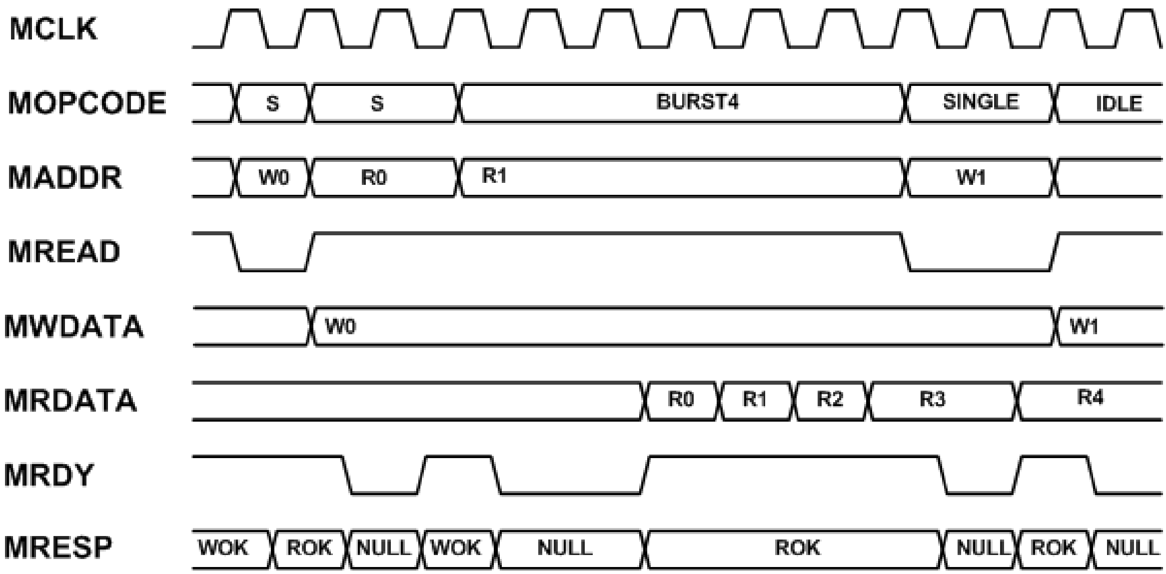
```
// Corresponding alternative version of the driver BFM drive task
task drive(bus_seq_item req, output bus_seq_item rsp);
    repeat (req.delay) begin // Delay between bus transactions
        @(posedge BUS.clk);
    end

    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 request with response data into the rsp
    $cast(rsp, req.clone()); // Clone the req
    if (req.read_not_write == 1) begin
        rsp.read_data = BUS.read_data; // Copy read data response
    end
    rsp.error = BUS.error; // Copy bus error response status

    BUS.valid <= 0; // End the pin level bus transaction
endtask: drive
```

```
// Corresponding version of the sequence body method
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);
        get_response(rsp);
        // The rsp handle points to the object that the driver has updated
with response data
        `uvm_info("seq_body", rsp.convert2string(), UVM_MEDIUM);
    end endtask:
body
```

For more information on this implementation style, especially how to initialize the response item, see the section on the get-put use model in the Driver/Sequence API article.

Pipelined Protocols

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 bus 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 with 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. While the data phase for one transfer is taking place on the second stage of the pipeline, the address phase for the next transfer can take place on the first stage of the pipeline. Other protocols such as OCP use more phases.



Pipelined bus protocol – Timing diagram

A pipelined protocol has the potential to increase the bandwidth of a system, since it increases the number of transfers that can take place over a given number of clock cycles, provided the pipeline is kept full. 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 driver, composed of a BFM-proxy pair in the dual domain testbench, could be modelled with a simple bidirectional style whereby the sequence sends a sequence item to the driver proxy which unblocks the sequence when the corresponding bus transaction has completed. 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 use model, a number of design considerations need to be taken into account in order to support fully pipelined transfers:

- **Driver Implementation** - The driver (BFM or proxy) needs to have multiple threads running, where each thread takes a sequence item through one of the pipeline stages.
- **Keeping the pipeline full** - The driver needs to unblock the sequencer in a timely manner to get the next sequence item so that the pipeline can be kept full.
- **Sequence Implementation** - The sequence needs to have separate stimulus request and response threads, with the former continually generating new bus transactions for the driver to keep the pipeline full.

Recommended Implementation Pattern Using Get and Put

A simple way to model a pipelined protocol with a sequence and 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 be able to process multiple sequence_items concurrently. A simple example to achieve this is by a driver proxy continually taking a sequence_item and starting the associated bus transfer - pipeline stage 1 - then promptly reiterating for the next item *before* full completion of the initiated transfer - pipeline stage 2. Each stage in the pipeline requires a separate thread, in this case one thread to get() a next bus transaction and execute its address/command portion, and the second thread to complete its data portion (in parallel with the address/command portion of the subsequent bus transaction).

To ensure that no more than one bus transaction enters each pipeline stage at a time, several complementary techniques exist to implement mutual exclusion. For instance, exclusive access to the address/command phase of a bus transaction can be accomplished through a blocking call from the driver proxy (task begin_transfer(...)) upon getting the next sequence_item to execute this first phase. Subsequent exclusive access to the concurrently threaded data phase of the bus transaction triggered by completion of the address/command phase can then be guaranteed using a simple semaphore-based locking mechanism. The semaphore is grabbed when available prior to executing the data phase, and released again upon phase completion. Note that at the end of this final phase of the bus transaction, a response must be "put()" back to the originating sequence to satisfy the UVM sequencer-driver handshake protocol (function end_transfer(...)).

The following code example shows the two-stage pipeline described above to further illustrate these principles.

```
class mbus_pipelined_driver extends uvm_driver #(mbus_seq_item);

`uvm_component_utils(mbus_pipelined_driver)

local virtual mbus_pipelined_driver_bfm m_bfm;

local mbus_seq_item pipeline [$];

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

function void set_bfm(virtual mbus_pipelined_driver_bfm bfm);
  m_bfm = bfm;
  m_bfm.proxy = this;
endfunction: set_bfm

task run_phase(uvm_phase phase);
  m_bfm.wait_for_reset();

  do_pipelined_transfers();
endtask

task do_pipelined_transfers();
```

```
    mbus_seq_item req;

    forever begin
        seq_item_port.get(req);
        accept_tr(req, $time);
        void'(begin_tr(req, "pipelined_driver"));

        // This blocking call performs the command phase of the request and
      then returns
        // right away before completing the data phase, thus allowing the
command phase
        // of the subsequent request (next loop iteration) to occur in
parallel with the
        // data phase of the current request, and so implementing the
pipeline
        m_bfm.begin_transfer(req);

        pipeline.push_back(req);
    end
endtask: do_pipelined_transfers

// Function to complete the sequence item - driver handshake back to
the sequence,
// decoupled from the point of the originating request
function void end_transfer(mbus_seq_item req); mbus_seq_item rsp
    = pipeline.pop_front(); rsp.copy(req);

    seq_item_port.put(rsp); // End of req item // put_response(rsp); is
actually a function instead of task
    end_tr(rsp);
endfunction: end_transfer

endclass: mbus_pipelined_driver

interface mbus_pipelined_driver_bfm (mbus_if MBUS); import

mbus_pipelined_agent_pkg::*; mbus_pipelined_driver proxy;

mbus_seq_item current_tr;

event do_data_phase;

task begin_transfer(mbus_seq_item req);
    command_phase(req);
```

```
    pipeline_lock_get(); // Start of data phase: grab semaphore

    current_tr = req;
    ->do_data_phase;
endtask: begin_transfer

always begin
    @do_data_phase;

    @(posedge MBUS.MCLK);

    data_phase(current_tr);
    proxy.end_transfer(current_tr);

    pipeline_lock_put(); // End of data phase: release semaphore
end

task wait_for_reset();
    ...
endtask

task command_phase(mbus_seq_item req);
    MBUS.MADDR <= req.MADDR;
    MBUS.MREAD <= req.MREAD;
    MBUS.MOPCODE <= req.MOPCODE;
    @(posedge MBUS.MCLK);
    ...
endtask: command_phase

task data_phase(mbus_seq_item req);
    ...
    if (req.MREAD == 1) begin
        req.MRDATA = MBUS.MRDATA;
    end
endtask: data_phase

bit pipeline_lock = 0;

task pipeline_lock_get();
    while (pipeline_lock) @(posedge MBUS.MCLK);
    pipeline_lock = 1;
endtask: pipeline_lock_get

function void pipeline_lock_put(); pipeline_lock
    = 0;
endfunction: pipeline_lock_put
```

```
endinterface: mbus_pipelined_driver_bfm
```

Sequence Implementation

Non-pipelined Accesses

Often non-pipelined transfers are required because a typical bus fabric is mimicking what a software program does, namely accessing individual 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. For a non-pipelined UVM sequence that would work with the pipelined driver, the sequence body() method would call start_item(), finish_item() and get_response() methods in order, as illustrated in the code example below. The get_response() method blocks until the driver proxy provides a response using its put() method at the end of the bus transaction.

```
// This sequence shows how a series of non-pipelined 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_nonpipelined_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_nonpipelined_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);
    `uvm_info("", $sformatf("write (i = %0d) of %h at %h", i,
req.MWDATA, req.MADDR), UVM_MEDIUM);
  end

  foreach (addr[i]) begin
    start_item(req);
    req.MADDR = addr[i];
    req.MREAD = 1;
```

```
    finish_item(req);
    get_response(req);
    `uvm_info("", $sformatf("read (i = %0d) of %h at %h", i,
req.MRDATA, req.MADDR), UVM_MEDIUM);
    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_nonpipelined_seq
```

Note: This example sequence has data checking built-in, merely to demonstrate how the read data value can be used. Normally, the specific type of checking required would be done using a separate scoreboard.

Pipelined Accesses

Pipelined accesses are primarily used to stress-test the bus and require a different approach in the sequence. A pipelined sequence needs to have separate threads for generating the request sequence_items and handling the response sequence_items.

The generation loop will block on each finish_item() call until one of the threads in the driver proxy completes a get() call. Once the loop is unblocked it needs to generate a new sequence_item to provide something for the next driver proxy thread to get(). Note that a new request sequence_item needs to be generated in each iteration of the loop; if only one request item handle were used, the driver proxy would be attempting to execute the request while the sequence is modifying its fields.

There is no response handling in the given example sequence. The assumption here is that data-checking will be done by a separate scoreboard as noted above. However, with the get() and put() driver implementation, there is a response FIFO in the sequence that must be managed. In the example, the response_handler is enabled using the use_response_handler() method, and the response_handler function is then called each time a response becomes available, keeping the sequence response FIFO empty. In this case the response_handler keeps count of the number of transactions to ensure that the sequence only exits upon completion of the final transaction.

```
// This is a pipelined version of the previous sequence with no
blocking call to get_response().
// There is no intent to check the data, which 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
prematurely

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);
    `uvm_info("", $sformatf("write (i = %0d) of %h at %h", i,
req.MWDATA, req.MADDR), UVM_MEDIUM);
  end

  foreach (addr[i]) begin
    start_item(req);
    req.MADDR = addr[i];
    req.MREAD = 1;
    finish_item(req);
    `uvm_info("", $sformatf("read (i = %0d) of %h at %h", i,
req.MRDATA, req.MADDR), UVM_MEDIUM);
  end

  wait(count == 20); // Do not end the sequence until the last req item
 is complete
endtask: body

// The response_handler function serves 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, the response_handler function should be extended.

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 for the driver proxy to signal to the sequence that it has completed a specific transaction phase. In the example below, a uvm_event_pool is used for the events along with two methods 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
function void trigger(string evnt); uvm_event e =
   events.get(evnt); e.trigger();
endfunction: trigger

// do_copy(), do_compare etc
```

```
endclass: mbus_seq_item
```

Driver Signalling Completion Using sequence_item Events

The driver proxy is almost identical to the get-put implementation above. Instead of using the put() method, it triggers the phase-done notification events in the sequence_item to signal phase completion back to the sequence, and make response information available via the sequence_item handle.

```
class mbus_pipelined_driver extends uvm_driver #(mbus_seq_item);

...

task do_pipelined_transfers();
   mbus_seq_item req;

   forever begin
      seq_item_port.get(req);
      accept_tr(req, $time);
      void'(begin_tr(req, "pipelined_driver"));

      // This blocking call performs the command phase of the request and
     then returns
      // right away before completing the data phase, thus allowing the
command phase
      // of the subsequent request (next loop iteration) to occur in
parallel with the
      // data phase of the current request, and so implementing the
pipeline
      m_bfm.begin_transfer(req);

      pipeline.push_back(req);

      req.trigger("CMD_DONE"); // Signal CMD_DONE at this end of command
phase
   end
endtask: do_pipelined_transfers

// Function to complete the sequence item - driver handshake back to
the sequence,
// decoupled from the point of the originating request
function void end_transfer(mbus_seq_item req); mbus_seq_item rsp
   = pipeline.pop_front(); rsp.copy(req);

   rsp.trigger("DATA_DONE"); // Signal DATA_DONE at this end of data
phase
   end_tr(rsp);
endfunction: end_transfer
```

```
endclass: mbus_pipelined_driver
```

Non-pipelined Access Sequences

Non-pipelined accesses are made from sequences which block, after completing the finish_item() call, by waiting for the event that notifies data phase completion. 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 completion event occurs.

```
// Task: finish_item
// Calls super.finish_item() but then also waits for the sequence
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 );
  super.finish_item( item , set_priority );   // The "normal"
finish_item()

  item.wait_trigger("DATA_DONE");   // Wait for the data phase to
complete
endtask: finish_item
```

As in the previous example of an non-pipelined sequence, the code example below has a data integrity check 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 data

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);
```

```
    req.wait_trigger("DATA_DONE");
    `uvm_info("", $sformatf("write (i = %0d) of %h at %h", i,
req.MWDATA, req.MADDR), UVM_MEDIUM);
  end

  foreach (addr[i]) begin
    start_item(req);
    req.MADDR = addr[i];
    req.MREAD = 1;
    finish_item(req);
    req.wait_trigger("DATA_DONE");
    `uvm_info("", $sformatf("read (i = %0d) of %h at %h", i,
req.MRDATA, req.MADDR), UVM_MEDIUM);
    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

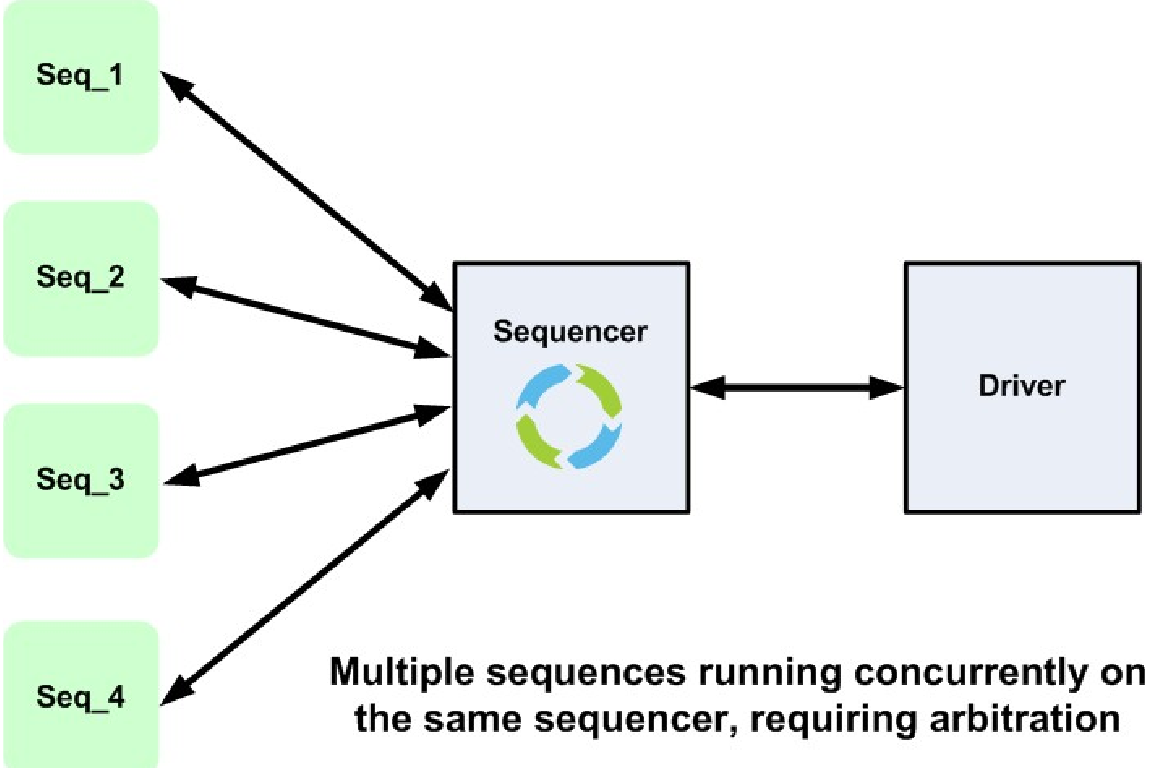
The pipelined access sequence does not wait for the data phase completion event notification to generate the next sequence item. Unlike for the get-put driver model, there is no need to manage the response FIFO, and so this implementation is a little more straightforward in this respect.

```
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);
    `uvm_info("", $sformatf("write (i = %0d) of %h at %h", i,
```

```
req.MWDATA, req.MADDR), UVM_MEDIUM);
  end

  foreach (addr[i]) begin
    start_item(req);
    req.MADDR = addr[i];
    req.MREAD = 1;
    finish_item(req);
    `uvm_info("", $sformatf("read (i = %0d) of %h at %h", i,
req.MRDATA, req.MADDR), UVM_MEDIUM);
  end
endtask: body

endclass: mbus_pipelined_seq
```

Arbitrating Between Sequences

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
               #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.

NOTE:

From UVM1.2 onwards, including UVM 1800.2, the names of the enum elements were changed to add the "UVM_" prefix.

For instance UVM_SEQ_ARB_FIFO was known as SEQ_ARB_FIFO in UVM versions prior to UVM1.2.

UVM_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: [Sqr Arb selected:] UVM_SEQ_ARB_FIFO
# UVM_INFO @ 1: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: [RCVD] Totals: SEQ_1:1 SEQ_2:1 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 21: [RCVD] Totals: SEQ_1:2 SEQ_2:1 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 31: [RCVD] Totals: SEQ_1:2 SEQ_2:1 SEQ_3:1 SEQ_4:0
# UVM_INFO @ 41: [RCVD] Totals: SEQ_1:2 SEQ_2:1 SEQ_3:1 SEQ_4:1
# UVM_INFO @ 51: [RCVD] Totals: SEQ_1:2 SEQ_2:2 SEQ_3:1 SEQ_4:1
# UVM_INFO @ 61: [RCVD] Totals: SEQ_1:3 SEQ_2:2 SEQ_3:1 SEQ_4:1
# UVM_INFO @ 71: [RCVD] Totals: SEQ_1:3 SEQ_2:2 SEQ_3:2 SEQ_4:1
# UVM_INFO @ 81: [RCVD] Totals: SEQ_1:3 SEQ_2:2 SEQ_3:2 SEQ_4:2
# UVM_INFO @ 91: [RCVD] Totals: SEQ_1:3 SEQ_2:3 SEQ_3:2 SEQ_4:2
# UVM_INFO @ 101: [RCVD] Totals: SEQ_1:4 SEQ_2:3 SEQ_3:2 SEQ_4:2
# UVM_INFO @ 111: [RCVD] Totals: SEQ_1:4 SEQ_2:3 SEQ_3:3 SEQ_4:2
# UVM_INFO @ 121: [RCVD] Totals: SEQ_1:4 SEQ_2:3 SEQ_3:3 SEQ_4:3
# UVM_INFO @ 131: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:3 SEQ_4:3
# UVM_INFO @ 141: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:4
```

UVM_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 more often than ones from lower priority seq_3 and seq_4. The resultant log file illustrates this:

```
# UVM_INFO @ 0: [Sqr Arb selected:] UVM_SEQ_ARB_WEIGHTED
# UVM_INFO @ 1: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:1
# UVM_INFO @ 21: [RCVD] Totals: SEQ_1:1 SEQ_2:1 SEQ_3:0 SEQ_4:1
# UVM_INFO @ 31: [RCVD] Totals: SEQ_1:1 SEQ_2:2 SEQ_3:0 SEQ_4:1
# UVM_INFO @ 41: [RCVD] Totals: SEQ_1:1 SEQ_2:3 SEQ_3:0 SEQ_4:1
# UVM_INFO @ 51: [RCVD] Totals: SEQ_1:1 SEQ_2:3 SEQ_3:1 SEQ_4:1
# UVM_INFO @ 61: [RCVD] Totals: SEQ_1:1 SEQ_2:4 SEQ_3:1 SEQ_4:1
# UVM_INFO @ 71: [RCVD] Totals: SEQ_1:2 SEQ_2:4 SEQ_3:1 SEQ_4:1
# UVM_INFO @ 81: [RCVD] Totals: SEQ_1:3 SEQ_2:4 SEQ_3:1 SEQ_4:1
# UVM_INFO @ 91: [RCVD] Totals: SEQ_1:3 SEQ_2:4 SEQ_3:1 SEQ_4:2
# UVM_INFO @ 101: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:1 SEQ_4:2
# UVM_INFO @ 111: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:2 SEQ_4:2
# UVM_INFO @ 121: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:3 SEQ_4:2
# UVM_INFO @ 131: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:3 SEQ_4:3
# UVM_INFO @ 141: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:4
```

UVM_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: [Sqr Arb selected:] UVM_SEQ_ARB_RANDOM
# UVM_INFO @ 1: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:1
# UVM_INFO @ 21: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:2
# UVM_INFO @ 31: [RCVD] Totals: SEQ_1:1 SEQ_2:1 SEQ_3:0 SEQ_4:2
# UVM_INFO @ 41: [RCVD] Totals: SEQ_1:1 SEQ_2:1 SEQ_3:1 SEQ_4:2
# UVM_INFO @ 51: [RCVD] Totals: SEQ_1:1 SEQ_2:2 SEQ_3:1 SEQ_4:2
# UVM_INFO @ 61: [RCVD] Totals: SEQ_1:2 SEQ_2:2 SEQ_3:1 SEQ_4:2
# UVM_INFO @ 71: [RCVD] Totals: SEQ_1:3 SEQ_2:2 SEQ_3:1 SEQ_4:2
# UVM_INFO @ 81: [RCVD] Totals: SEQ_1:3 SEQ_2:2 SEQ_3:1 SEQ_4:3
# UVM_INFO @ 91: [RCVD] Totals: SEQ_1:3 SEQ_2:3 SEQ_3:1 SEQ_4:3
# UVM_INFO @ 101: [RCVD] Totals: SEQ_1:3 SEQ_2:4 SEQ_3:1 SEQ_4:3
# UVM_INFO @ 111: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:1 SEQ_4:3
# UVM_INFO @ 121: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:2 SEQ_4:3
# UVM_INFO @ 131: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:2 SEQ_4:4
# UVM_INFO @ 141: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:3 SEQ_4:4
# UVM_INFO @ 151: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:4
```

UVM_SEQ_ARB_STRICT_FIFO

The UVM_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: [Sequencer Arbitration selected:]
UVM_SEQ_ARB_STRICT_FIFO
# UVM_INFO @ 1: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: [RCVD] Totals: SEQ_1:1 SEQ_2:1 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 21: [RCVD] Totals: SEQ_1:2 SEQ_2:1 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 31: [RCVD] Totals: SEQ_1:2 SEQ_2:2 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 41: [RCVD] Totals: SEQ_1:3 SEQ_2:2 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 51: [RCVD] Totals: SEQ_1:3 SEQ_2:3 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 61: [RCVD] Totals: SEQ_1:4 SEQ_2:3 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 71: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 81: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:1 SEQ_4:0
# UVM_INFO @ 91: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:2 SEQ_4:0
# UVM_INFO @ 101: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:3 SEQ_4:0
# UVM_INFO @ 111: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:0
# UVM_INFO @ 121: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:1
# UVM_INFO @ 131: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:2
```

```
# UVM_INFO @ 141: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:4
```

UVM_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: [Sqr Arb selected:] UVM_SEQ_ARB_STRICT_RANDOM
# UVM_INFO @ 1: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: [RCVD] Totals: SEQ_1:2 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 21: [RCVD] Totals: SEQ_1:3 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 31: [RCVD] Totals: SEQ_1:3 SEQ_2:1 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 41: [RCVD] Totals: SEQ_1:3 SEQ_2:2 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 51: [RCVD] Totals: SEQ_1:4 SEQ_2:2 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 61: [RCVD] Totals: SEQ_1:4 SEQ_2:3 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 71: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 81: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:1 SEQ_4:0
# UVM_INFO @ 91: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:2 SEQ_4:0
# UVM_INFO @ 101: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:3 SEQ_4:0
# UVM_INFO @ 111: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:0
# UVM_INFO @ 121: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:1
# UVM_INFO @ 131: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:2
# UVM_INFO @ 141: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:3
# UVM_INFO @ 151: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:4
```

UVM_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 sequencers 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 int user_priority_arbitration(int avail_sequences[$]);
 int priority[] = new[avail_sequences.size]
 int 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: [Sqr Arb selected:] UVM_SEQ_ARB_USER
# UVM_INFO @ 1: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:0
# UVM_INFO @ 11: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:1
# UVM_INFO @ 21: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:2
# UVM_INFO @ 31: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:3
# UVM_INFO @ 41: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:0 SEQ_4:4
# UVM_INFO @ 51: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:1 SEQ_4:4
# UVM_INFO @ 61: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:2 SEQ_4:4
# UVM_INFO @ 71: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:3 SEQ_4:4
# UVM_INFO @ 81: [RCVD] Totals: SEQ_1:1 SEQ_2:0 SEQ_3:4 SEQ_4:4
```

```
# UVM_INFO @ 91: [RCVD] Totals: SEQ_1:2 SEQ_2:0 SEQ_3:4 SEQ_4:4
# UVM_INFO @ 101: [RCVD] Totals: SEQ_1:3 SEQ_2:0 SEQ_3:4 SEQ_4:4
# UVM_INFO @ 111: [RCVD] Totals: SEQ_1:4 SEQ_2:0 SEQ_3:4 SEQ_4:4
# UVM_INFO @ 121: [RCVD] Totals: SEQ_1:4 SEQ_2:1 SEQ_3:4 SEQ_4:4
# UVM_INFO @ 131: [RCVD] Totals: SEQ_1:4 SEQ_2:2 SEQ_3:4 SEQ_4:4
# UVM_INFO @ 141: [RCVD] Totals: SEQ_1:4 SEQ_2:3 SEQ_3:4 SEQ_4:4
# UVM_INFO @ 151: [RCVD] Totals: SEQ_1:4 SEQ_2:4 SEQ_3:4 SEQ_4:4
```

Sequence 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 UVM_SEQ_ARB_WEIGHTED, UVM_SEQ_ARB_STRICT_FIFO, and UVM_SEQ_ARB_STRICT_RANDOM arbitration mechanisms, (and possibly the UVM_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(UVM_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.

Locking or Grabbing a Sequencer

There are a number of modeling 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 accommodate this requirement, the uvm_sequencer has a locking mechanism which is implemented using two calls - lock() and grab(). In terms of modeling, a lock might be used to model a prioritized 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

  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 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:

(Note that there has been some editing of the transcript to ensure that it renders reasonably in pdf)

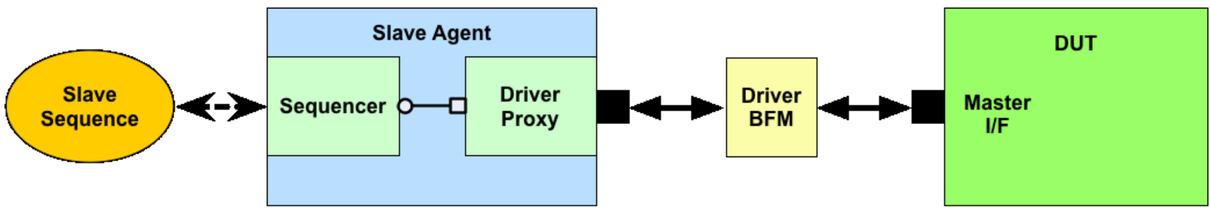
```
UVM_INFO @ 0: reporter [RNTST] Running test arb_test...
# UVM_INFO @ 0: [Sequencer Arbitration selected:] SEQ_ARB_FIFO
# UVM_INFO @ 1:     [RCVD] Type: 1 S1:1 S2:0 S3:0 S4:0 GB:0 LK:0
# UVM_INFO @ 10: [grab_seq] This seq is NOT blocked by an existing lock
# UVM_INFO @ 11:    [RCVD] Type: 5 S1:1 S2:0 S3:0 S4:0 GB:1 LK:0
# UVM_INFO @ 12: [is_blocked] This seq is blocked by a lock or a grab
# UVM_INFO @ 21:    [RCVD] Type: 5 S1:1 S2:0 S3:0 S4:0 GB:2 LK:0
# UVM_INFO @ 31:    [RCVD] Type: 5 S1:1 S2:0 S3:0 S4:0 GB:3 LK:0
# UVM_INFO @ 41:    [RCVD] Type: 5 S1:1 S2:0 S3:0 S4:0 GB:4 LK:0
# UVM_INFO @ 51:    [RCVD] Type: 2 S1:1 S2:1 S3:0 S4:0 GB:4 LK:0
# UVM_INFO @ 61:    [RCVD] Type: 3 S1:1 S2:1 S3:1 S4:0 GB:4 LK:0
# UVM_INFO @ 71:    [RCVD] Type: 4 S1:1 S2:1 S3:1 S4:1 GB:4 LK:0
# UVM_INFO @ 81:    [RCVD] Type: 1 S1:2 S2:1 S3:1 S4:1 GB:4 LK:0
# UVM_INFO @ 91:    [RCVD] Type: 2 S1:2 S2:2 S3:1 S4:1 GB:4 LK:0
# UVM_INFO @ 101:   [RCVD] Type: 3 S1:2 S2:2 S3:2 S4:1 GB:4 LK:0
# UVM_INFO @ 111:   [RCVD] Type: 4 S1:2 S2:2 S3:2 S4:2 GB:4 LK:0
# UVM_INFO @ 121:   [RCVD] Type: 1 S1:3 S2:2 S3:2 S4:2 GB:4 LK:0
# UVM_INFO @ 131:   [RCVD] Type: 2 S1:3 S2:3 S3:2 S4:2 GB:4 LK:0
# UVM_INFO @ 141:   [RCVD] Type: 3 S1:3 S2:3 S3:3 S4:2 GB:4 LK:0
# UVM_INFO @ 151:   [RCVD] Type: 4 S1:3 S2:3 S3:3 S4:3 GB:4 LK:0
# UVM_INFO @ 161:[lock_seq] This seq is NOT blocked by an existing lock
# UVM_INFO @ 161:   [RCVD] Type: 6 S1:3 S2:3 S3:3 S4:3 GB:4 LK:1
# UVM_INFO @ 171:   [RCVD] Type: 6 S1:3 S2:3 S3:3 S4:3 GB:4 LK:2
# UVM_INFO @ 181:[grab_seq] This seq IS BLOCKED by an existing lock
# UVM_INFO @ 181:   [RCVD] Type: 6 S1:3 S2:3 S3:3 S4:3 GB:4 LK:3
# UVM_INFO @ 191:   [RCVD] Type: 6 S1:3 S2:3 S3:3 S4:3 GB:4 LK:4
# UVM_INFO @ 201:   [RCVD] Type: 5 S1:3 S2:3 S3:3 S4:3 GB:5 LK:4
# UVM_INFO @ 211:   [RCVD] Type: 5 S1:3 S2:3 S3:3 S4:3 GB:6 LK:4
# UVM_INFO @ 221:   [RCVD] Type: 5 S1:3 S2:3 S3:3 S4:3 GB:7 LK:4
# UVM_INFO @ 231:   [RCVD] Type: 5 S1:3 S2:3 S3:3 S4:3 GB:8 LK:4
# UVM_INFO @ 241:   [RCVD] Type: 1 S1:4 S2:3 S3:3 S4:3 GB:8 LK:4
# UVM_INFO @ 251:   [RCVD] Type: 2 S1:4 S2:4 S3:3 S4:3 GB:8 LK:4
# UVM_INFO @ 261:   [RCVD] Type: 3 S1:4 S2:4 S3:4 S4:3 GB:8 LK:4
# UVM_INFO @ 271:   [RCVD] Type: 1 S1:5 S2:4 S3:4 S4:3 GB:8 LK:4
# UVM_INFO @ 281:   [RCVD] Type: 2 S1:5 S2:5 S3:4 S4:3 GB:8 LK:4
# UVM_INFO @ 291:   [RCVD] Type: 3 S1:5 S2:5 S3:5 S4:3 GB:8 LK:4
# UVM_INFO @ 301:   [RCVD] Type: 1 S1:6 S2:5 S3:5 S4:3 GB:8 LK:4
# UVM_INFO @ 311:   [RCVD] Type: 2 S1:6 S2:6 S3:5 S4:3 GB:8 LK:4
# UVM_INFO @ 321:   [RCVD] Type: 3 S1:6 S2:6 S3:6 S4:3 GB:8 LK:4
# UVM_INFO @ 331:   [RCVD] Type: 1 S1:7 S2:6 S3:6 S4:3 GB:8 LK:4
```

```
# UVM_INFO @  341:    [RCVD]  Type:  2   S1:7   S2:7    S3:6   S4:3   GB:8  LK:4
# UVM_INFO @  351:    [RCVD]  Type:  3   S1:7   S2:7    S3:7   S4:3   GB:8  LK:4
# UVM_INFO @  361:    [RCVD]  Type:  1   S1:8   S2:7    S3:7   S4:3   GB:8  LK:4
# UVM_INFO @  371:    [RCVD]  Type:  2   S1:8   S2:8    S3:7   S4:3   GB:8  LK:4
# UVM_INFO @  381:    [RCVD]  Type:  3   S1:8   S2:8    S3:8   S4:3   GB:8  LK:4
# UVM_INFO @  391:    [RCVD]  Type:  1   S1:9   S2:8    S3:8   S4:3   GB:8  LK:4
# UVM_INFO @  401:    [RCVD]  Type:  2   S1:9   S2:9    S3:8   S4:3   GB:8  LK:4
# UVM_INFO @  411:    [RCVD]  Type:  3   S1:9   S2:9    S3:9   S4:3   GB:8  LK:4
# UVM_INFO @ 421:    [RCVD] Type: 1 S1:10 S2:9 S3:9 S4:3 GB:8 LK:4
# UVM_INFO @ 431:    [RCVD] Type: 2 S1:10 S2:10 S3:9 S4:3 GB:8 LK:4
# UVM_INFO @ 441:    [RCVD] Type: 3 S1:10 S2:10 S3:10 S4:3 GB:8 LK:4
```

Slave Sequences (Responders)

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 Slave Agent responds to requests on the master interface of the DUT

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;

  m_cfg.wait_for_reset();
  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;

  m_bfm.reset();

  forever begin
    // Setup Phase
    seq_item_port.get_next_item(req);

    // Setup phase activity
    m_bfm.setup_phase(req);

    seq_item_port.item_done();

    // Access Phase
    seq_item_port.get_next_item(rsp);

    // Access phase activity
    m_bfm.access_phase(req, rsp);

    seq_item_port.item_done();
   end

endtask: run_phase
```

Example	Download Link
Complete APB3 Slave Agent	

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 parameterized 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 and calls the BFM.

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;

  m_bfm.reset();

  forever begin
    // Setup Phase
    seq_item_port.get_next_item(item);
    if ($cast(req, item)) begin
       m_bfm.setup_phase(req);
    end else begin
      `uvm_error("CASTFAIL", "The received sequence item is not a
request seq_item");
    end
    seq_item_port.item_done();

    // Access Phase
    seq_item_port.get_next_item(item);
    if ($cast(rsp, item)) begin
      m_bfm.access_phase(rsp);
    end else begin
      `uvm_error("CASTFAIL", "The received sequence item is not a
response seq_item");
    end
    seq_item_port.item_done();
  end
endtask: run_phase
```

Example	Download Link
Complete APB3 Slave Agent using multiple sequence items	

Wait for a Signal

In the general case, synchronizing to hardware events is taken care of in UVM testbenches by drivers (proxy & BFM) and monitors (proxy & BFM). However, there are some cases where it is useful for a sequence or a component to synchronize 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 BFM handle (usually a configuration object) that call a task in the BFM that blocks until a hardware event occurs. 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.

BFM Methods

Since BFMs are the only items which are allowed to access pins directly, the BFMs have hardware synchronization methods added to them which can then be called by code in a testbench such as configuration objects. Examples of hardware synchronization methods include:

- wait_for_clock()
- wait_for_reset()
- wait_for_interrupt()
- interrupt_cleared()

These hardware synchronization methods are declared explicitly as automatic because static entities such as interfaces have implicitly static tasks. Since multiple threads/locations in a testbench way want to wait for a number of clock cycles or wait for a reset signal, we need to be able to invoke the tasks multiple times without overwriting the values used for each invocation.

In general the monitor BFM should be targeted for adding the extra methods since the monitor BFM would be present in both active and passive modes.

An example BFM:

```
interface bidirect_bus_monitor_bfm (bus_if BUS);

//
// Task: wait_for_clock
//
// This method waits for n clock cycles.
task automatic wait_for_clock( int n = 1 );
 repeat( n ) @( posedge BUS.clk );
endtask : wait_for_clock

//
// Task: wait_for_reset
//
// This method waits for the end of reset.
task automatic wait_for_reset();
 // Wait for reset to end
 @(posedge BUS.resetn);
endtask : wait_for_reset

//
```

```
// Task: wait_for_error
//
task automatic wait_for_error;
 @(posedge error);
endtask: wait_for_error

//
// Task: wait_for_no_error
//
task automatic wait_for_no_error;
 @(negedge error);
endtask: wait_for_no_error

endinterface : bidirect_bus_monitor_bfm
```

Additions to the configuration object

In order to support the use of a wait_for_interface_signal method, hardware synchronization methods have to be added to a configuration object. These hardware synchronization methods reference methods within the virtual interface BFM(s) which the configuration object contains handles for.

An example:

```
class bus_agent_config extends uvm_object;

`uvm_object_utils(bus_agent_config)

virtual bidirect_bus_monitor_bfm mon_bfm;  // This is the virtual
interface with the methods to wait on

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

//
// Task: wait_for_clock
//
// This method waits for n clock cycles.
task wait_for_clock( int n = 1 );
 mon_bfm.wait_for_clock(n);
endtask

//
// Task: wait_for_reset
//
// This method waits for the end of reset.
task wait_for_reset();
  mon_bfm.wait_for_reset();
endtask : wait_for_reset
```

```
//
// Task: wait_for_error
//
task wait_for_error();
  mon_bfm.wait_for_error();
endtask : wait_for_error

//
// Task: wait_for_no_error
//
task wait_for_no_error();
  mon_bfm.wait_for_no_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 (m_cfg == null)
    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 (m_cfg == null)
      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
```

Interrupt Sequences

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
- Prioritized 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");
```

```
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");
```

```
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 prioritization since each IRQ source is tested in order from IRQ0 to IRQ3.

Prioritised ISR Sequence

A less disruptive approach to implementing interrupt handling using sequences is to use sequence prioritization. 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.

Prioritizing ISR sequences also enables modeling of prioritized 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 prioritization 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
```

```
isr 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
  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
```

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
```

Stopping a Sequence

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.

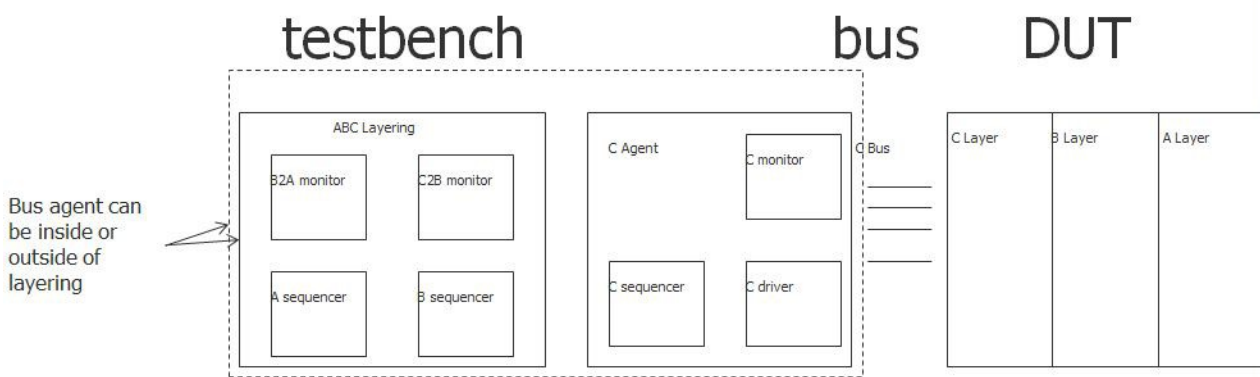
The methods are:

- <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.

Layering Sequences

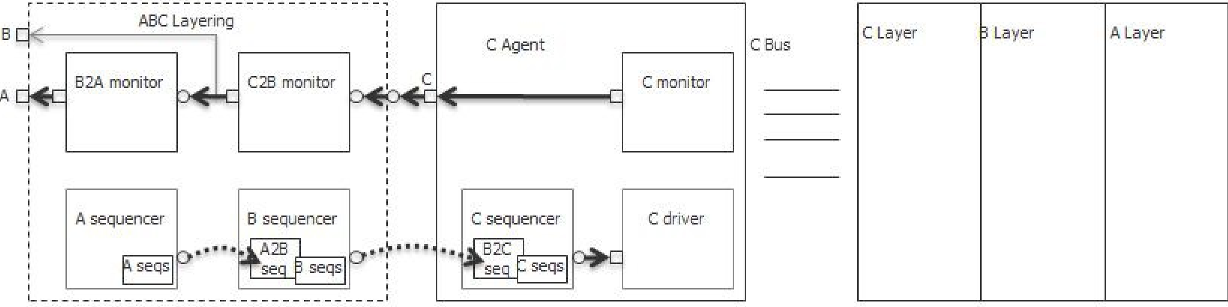
Many protocols have a hierarchical definition - for example, PCI express, USB 3.0, and MIPI LLI all have a Transaction Layer, a Transport Layer, and a Physical Layer. Sometimes we want to create a protocol independent layer on top of a standard protocol so that we can create protocol independent verification 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.



Data Flow

= via formal port to export communication

= via direct calls to upstream sequencer's export

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:

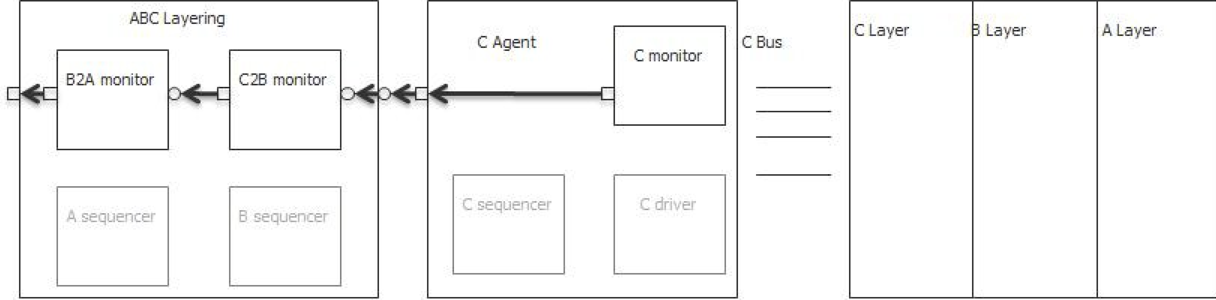
```
class ABC_layering extends uvm_component;
  `uvm_component_utils( ABC_layering )
```

```
  ...
  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
```

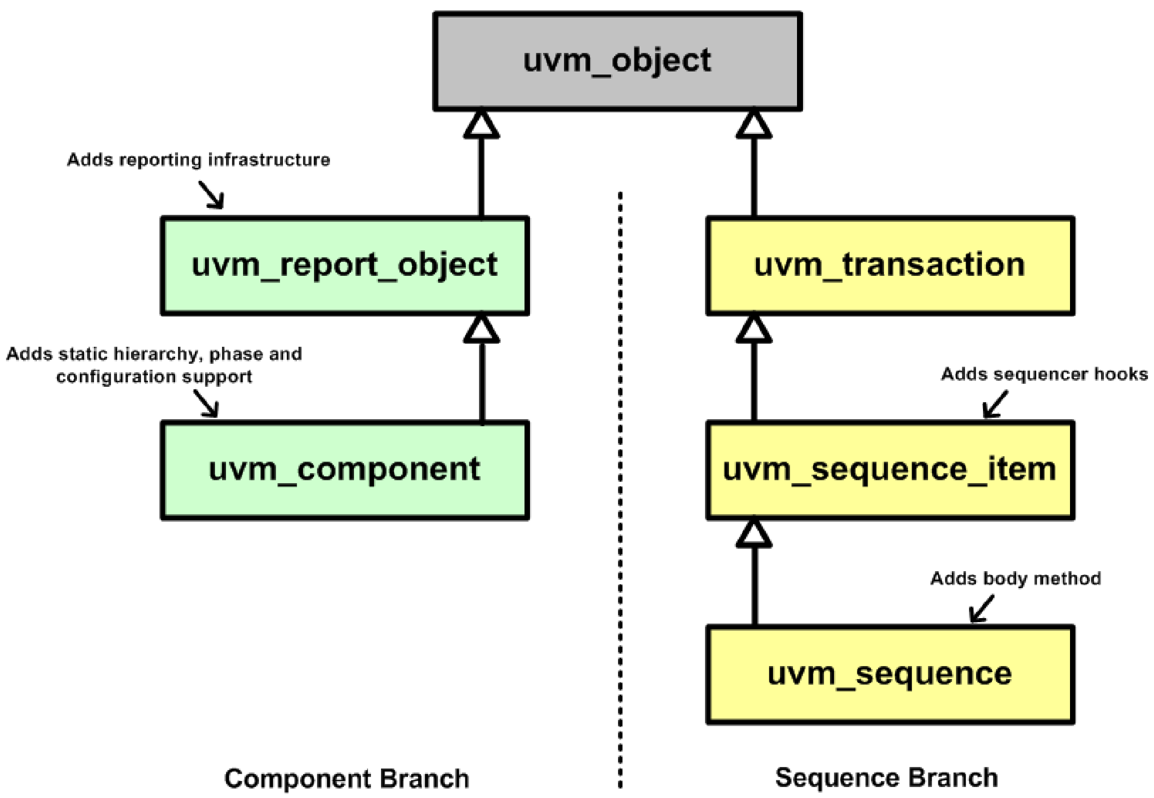
The reconstruction monitors are connected up in the normal way:

```
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
```

The UVM Messaging System

UVM Messaging

The UVM messaging system provides an infrastructure for printing messages in a consistent format from a UVM testbench. The messaging service is built into all UVM components, since they are extended from the uvm_report_handler as shown in the UVM class inheritance diagram. Messaging can also be used from UVM objects such as sequence_items and sequences.



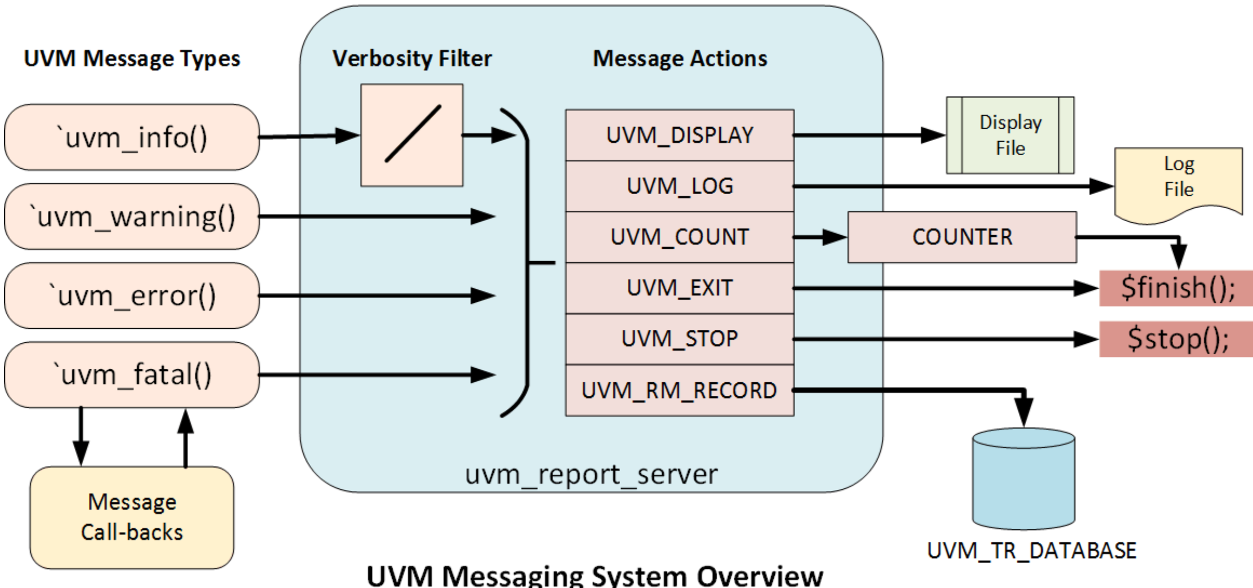
There are four message types supported by the UVM messaging system:

Message Type	Suggested useage
UVM_INFO	Should be used for informative messages. Its verbosity can be controlled
UVM_WARNING	Should be used for warnings to the user
UVM_ERROR	Should be used to flag errors to the user
UVM_FATAL	Should be used for errors that will prevent the test from doing anything useful. Also results in an exit from the simulation

The system has a verbosity control capability that allows the suppression of UVM_INFO messages. There is no verbosity control for UVM_WARNING, UVM_ERROR and UVM_FATAL messages.

UVM messages can have actions associated with them, and these allow control of where the message is sent; whether the message causes the simulation to stop or finish; or whether the message is stored in the UVM transaction database. For more detail about the features of the messaging system, see the section on using UVM messaging.

In addition to UVM based messaging API calls, the verbosity of the message reporting system can also be controlled, to a lesser extent, from the simulation command line.



UVM Messaging System Overview

A message can be intercepted by a report catcher call-back, allowing its content, type and severity to be changed. This feature is useful in some circumstances, for instance to demote an error to a warning, but should be used with care.

At the end of the simulation, the message server can be interrogated to see how many messages of a particular type have been issued. This is particularly useful as a sanity check in regressions to ensure that no error or warning messages have been reported.

When messages are sent from a UVM component, the message includes the UVM testbench path to the component. When messages are sent from objects such as sequences, then the path part of the message uses the result of the get_full_name() method. For sequences this does not include the full sequence hierarchy unless you follow a particular approach to starting sequences. See the article on sequence messaging for more details.

Printing a message has an overhead associated with it since there will be string processing in the UVM library and there will be some form of File I/O overhead in the simulator. This means that a noisy testbench can also be a slow testbench, so it is worth considering how verbose your messaging output needs to bewhen you are coding a verification component or a sequence.

Using Messaging

Messaging macros

The recommended way to use the UVM messaging system is to use the message macros, since they automatically insert the file name and line number of the message source into the UVM message string which is useful for debugging. In the case of information messages, the expanded macro code also checks the verbosity setting for the message to determine whether it should be printed before any string processing takes place. This can improve testbench performance.

The macros are:

```
`uvm_fatal(“message_id”, “message_string”)
`uvm_error(“message_id”, “message_string”)
`uvm_warning(“message_id”, “message_string”)
`uvm_info(“message_id”, “message_string”, uvm_verbosity)
```

Where:

The **message_id** is a string that can be used to identify the source of the message. It is used within the messaging system as a reference that allows you to control other aspects of the message behavior.

The **message_string** is what will be printed in the body of the message, it can be a simple string or it can be the result of a string formatting function such as $sformatf().

The **uvm_verbosity** is an enumerated type that defines the verbosity setting for the message.

Message Verbosity

The UVM has five levels of verbosity which are defined as an enum within the UVM package:

Verbosity Level	Effect on message
UVM_NONE	A message with this level of verbosity will always be printed regardless of the verbosity setting
UVM_LOW	A message with this level of verbosity will be printed if the verbosity is set to UVM_LOW, UVM_MEDIUM, UVM_HIGH or UVM_FULL
UVM_MEDIUM	A message with this level of verbosity will be printed is the verbosity is set to UVM_MEDIUM, UVM_HIGH or UVM_FULL. This is the default level.
UVM_HIGH	A message with this level of verbosity will be printed if the verbosity is set to UVM_HIGH or UVM_FULL
UVM_FULL	A message with this level of verbosity will be printed if the verbosity is set to UVM_FULL

The verbosity settings are a common source of confusion amongst users. The important thing to realize is that the name of the enum for a particular verbosity setting is related to the amount of verbosity allowed within a testbench or part of a testbench. The idea being that FULL relates to all possible messages being printed, and NONE to only essential messages being printed. For example, if an info message has a verbosity setting of UVM_HIGH, it will only be emitted if the verbosity setting is set to UVM_HIGH or UVM_FULL.

Each info message has to have a verbosity setting. Whether it is displayed or not is determined by the verbosity setting of the UVM testbench or the component from which it is generated.

Suggested uses of the settings are shown in the table below:

Verbosity Level	Suggested Usage
UVM_NONE	A message that will always be printed – e.g. Mandatory information such as Copyright or design/testbench configuration.
UVM_LOW	A message that should normally be printed, but could be filtered by a UVM_NONE setting. – e.g. Higher level messaging, Assertion messaging
UVM_MEDIUM	Debug level 1 – e.g. Phase has started/finished (Default verbosity level)
UVM_HIGH	Debug level 2 – e.g. Detailed transaction messages
UVM_FULL	Low level of detail debug messages

The UVM messaging verbosity is set to UVM_MEDIUM by default. The message verbosity level can be changed either for the whole UVM testbench or at a finer level of granularity.

Message IDs

The message ID field of a UVM message can be used as a way of grouping related messages and then controlling how they are handled. The ID string can take any format, and there is no limit as to how many can be used.

Message Actions

UVM_ACTION_e	Description
UVM_NONE	No action is taken – an alternative way to suppress a message.
UVM_DISPLAY	The message is directed to the simulation transcript
UVM_LOG	The message is directed to a specific log file
UVM_COUNT	The message increments an exit counter value
UVM_EXIT	The message causes the simulation to finish immediately
UVM_STOP	The message causes the simulation to stop and become interactive
UVM_RM_RECORD	The message is logged in the uvm_tr_database

These actions are encoded as a bit pattern and can be ORed together to create a hybrid action definition.

For any component, the actions for different message types can be changed either according to their severity or their id, or both. The various message action API calls are:

```
// Apply to a single level of the component hierarchy:
set_report_severity_action(uvm_severity severity, uvm_action action);
set_report_id_action(string id, uvm_action action);
set_report_severity_id_action(uvm_severity, string id, uvm_action
action);

// For instance:
// Any `uvm_info() messages from this_component with an id of
"this_agent" are sent to a log file and the transcript
this_component.set_report_severity_id_action(UVM_INFO, "this_agent",
UVM_LOG | UVM_DISPLAY);

// Apply to all components below the component in the hierarchy:
set_report_severity_action_hier(uvm_severity severity, uvm_action
action);
```

```
set_report_id_action_hier(string id, uvm_action action);
set_report_severity_id_action_hier(uvm_severity, string id, uvm_action
action);
```

Using UVM_LOG

In order to use the UVM_LOG action, you need to take responsibility for opening the log file before any messages are written to it, and for closing it at the end of the simulation. You also need to pass the file handle to the messaging system. Again, the log file can be set up as a default, or according to severity, id or both. The API calls are:

```
// Apply to a single level of the component hierarchy:
set_report_default_file(UVM_FILE file);
set_report_id_file(string id, UVM_FILE file);
set_report_severity_file(uvm_severity severity, UVM_FILE file);
set_report_severity_id_file(uvm_severity severity, string id, UVM_FILE
file);

// Apply to all components below the component in the hierarchy:
set_report_default_file_hier(UVM_FILE file);
set_report_id_file_hier(string id, UVM_FILE file);
set_report_severity_file_hier(uvm_severity severity, UVM_FILE file);
set_report_severity_id_file_hier(uvm_severity severity, string id,
UVM_FILE file);
```

The following code is an example of how the log file handle is created, applied and then how the log file is closed at the end of the simulation:

```
task run_phase(uvm_phase phase);

  // Create a file handle by opening a file:
  UVM_FILE green_log_fh = $fopen("green_messages.log");

  // Adding the log action for the "green_id":
  env.green.set_report_id_action("green_id", (UVM_DISPLAY | UVM_LOG));
  // Setting the file handle for the log action
  env.green.set_report_id_file("green_id", green_log_fh);


  phase.raise_objection(this);
  #1us;
  // Test code goes here
  phase.drop_objection(this);

  // Closing the log file at the end of test:
  $fclose(green_log_fh);
endtask
```

Note that an alternative place to close the file might be at the end of the report_phase(), especially if the report_phase() is going to write to the file!

Using UVM_COUNT

The UVM_COUNT action is useful for controlling how many errors occur before the UVM test is aborted, but in principle it can be used to control how many messages of any type allocated the UVM_COUNT action occur before the simulation exits. The maximum value of the counter is set using the set_max_quit_count() API call.

Using UVM_EXIT/UVM_STOP

The UVM_EXIT action means that if a message with this action is called, the simulation stops immediately. The uvm_fatal message class uses this action by default.

The UVM_STOP action makes the simulation stop and puts the simulator into interactive mode. This could be useful for debug.

Using UVM_RM_RECORD

The UVM_RM_RECORD action adds an entry to the UVM transaction database when the message is generated. This is a new option in the 1800.2 version of the UVM. How the transaction database works and how it is used is specific to a vendor implementation.

Default Message Action Settings

The four types of UVM message have default action settings as follows:

Message Type	Default Action Settings
UVM_INFO	UVM_DISPLAY
UVM_WARNING	UVM_DISPLAY
UVM_ERROR	UVM_DISPLAY|UVM_COUNT
UVM_FATAL	UVM_DISPLAY|UVM_EXIT

UVM Report Catcher

There are situations where you may need to change a message generated by the messaging system, and the uvm_report_catcher is built-in call-back mechanism for doing this. Typical applications might be to downgrade an error to a warning, or to modify a message. The report catcher can modify the severity, verbosity, id, action or the string content of a message before it is passed to the message server for printing.

The report catcher is implemented as an extended version of the uvm_callbacks object. Multiple report catchers can be registered and potentially each message will be processed by all the registered call-backs in the order in which they were registered. The report catcher's catch() method is called as the message is generated, and the implementation of the catch() method determines how the message is processed. If the catch() method returns a THROW value, then the message is passed onto other registered report catchers. If it returns a CAUGHT value then the message is passed to the report server. Inside the catch() method, the issue() call can be made which will immediately send the message to the report server.

Each uvm_report_catcher object needs to be constructed and then registered either as a global report catcher or as a report catcher for a group of components or a single component.

Report Catcher Examples

The first report catcher example defines a call-back that checks for a message with the "green_id" and if the message type is UVM_INFO, then it modifies the message. The same call-back also demotes any errors with a "green_id" to warnings. The call back returns a THROW value, which means that the message will be passed to the next report catcher call-back, if there is one.

```
// Example report catchers:
class message_mod extends uvm_report_catcher;
`uvm_object_utils(message_mod)

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

function action_e catch();
  string message;
  string id;

  if((get_id() == "green_id") & (get_severity() == UVM_INFO)) begin
    set_message("Message modified");
  end
  else if((get_id() == "green_id") & (get_severity() == UVM_ERROR))
begin
    set_severity(UVM_WARNING);
    set_message("This warning was an error");
  end

  return THROW;

endfunction
```

```
endclass
```

The following report catcher example is designed to catch a UVM_FATAL severity message with a "red_id" and demote it to an error. This type of report catcher might be useful in the early stages of debugging a testbench as a means of continuing past a fatal error.

```
class fatal_mod extends uvm_report_catcher;
`uvm_object_utils(fatal_mod)

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

function action_e catch();
  string message;
  string id;

  if((get_id() == "red_id") & (get_severity() == UVM_FATAL)) begin
    set_message("Something went very wrong but was demoted to error");
    set_severity(UVM_ERROR);
  end
  return THROW;

endfunction

endclass
```

The following UVM code shows how these two call-backs are declared, constructed and then registered. The message_mod call back is registered only against the env.green component, so it will only be called when an object in the env.green generates a message. The fatal_demoter call-back has its add() method type argument set to null, which means that it applies to all messages generated in the UVM testbench.

```
// A test class that uses them:
class message_test extends uvm_component;

message_env env;
message_mod mess_mod;
fatal_mod fatal_demoter;

function void build_phase(uvm_phase phase);
  env = message_env::type_id::create("env", this);
  mess_mod = new("mess_mod");
  fatal_demoter = new("fatal_demoter");
  // The Message_mod report catcher is only applied to the env.green
component:
  uvm_report_cb::add(env.green, mess_mod);
  // The fatal_mod report catcher is applied to all component messaging
  uvm_report_cb::add(null, fatal_demoter);
```

```
endfunction
```

At the end of the UVM simulation, the report server will generate a summary of all the fatal, error and warning messages that had their severity demoted using the report catcher.

Testing Message Status

At the end of a UVM simulation, the report server issues a messaging summary to the transcript of the simulation. This will detail the number of each type of message severity that has been generated by the messaging system. If the simulation has been run with no report modifications, then this summary can be parsed and used as part of determining whether the simulation passed or not. For instance, the test case might be deemed to have passed if the scoreboard issues a pass message and there are no UVM_ERRORS or UVM_WARNINGS.

However, there may be more complicated scenarios where messages have been downgraded, or a specific error should have been provoked a certain number of times during the test case. In this situation the report server can be queried during the report phase to determine whether the test behavior is as expected.

The report server has two methods that are useful for this:

```
function int get_id_count(string id);
function int get_severity_count(uvm_severity severity);
```

The following code is an example of how these API calls might help to determine whether a test has passed or failed:

```
    function void report_phase(uvm_phase phase);
      uvm_coreservice_t cs;
      uvm_report_server svr;
      cs = uvm_coreservice_t::get();
      svr = cs.get_report_server();

      if (svr.get_severity_count(UVM_FATAL) == 0 &&
          ((svr.get_id_count("ASSERT_PARITY_ERROR") == 5) &&
(svr.get_severity_count(UVM_ERROR) == 5))) begin
         $write("** UVM TEST PASSED **\n");
      end
      else begin
         $write("!! UVM TEST FAILED !!\n");
      end
    endfunction
```

In this example, the test is expected to provoke 5 parity errors, therefore if there not 5 UVM_ERROR messages reported with the "ASSERT_PARITY_ERROR" id, then something has gone wrong in the test.

Note: - This approach to determining whether a test has passed or not is only relevant if all the messages generated in the UVM testbench code have used the UVM messaging system. It should be used in combination with other checks, like checking the status of a scoreboard. For instance, you could have a scenario where there are no UVM_ERRORs reported, but the test has failed because no transactions were sent to the DUT. Also, if assertions are being used, it would be unusual for those to be using the UVM messaging system, so you typically need an alternative way of checking for any assertion errors.

Command-Line Verbosity Control

There are several UVM plusargs that can be used to control messaging verbosity, actions and severity from the command line:

PlusArg	Description
+UVM_VERBOSITY=<uvm_verbosity>	This will set the default verbosity of all the components created in the UVM testbench to the specified verbosity.
+uvm_set_verbosity=<component>,<id>,<verbosity>,<phase>	This will set the verbosity of a specific component for a given UVM phase
+uvm_set_verbosity=<component>,<id>,<verbosity>,time,<time>	This will set the verbosity of a specific component from a specific time in the simulation.
+uvm_set_action=<component>,<id>,<severity><action>	This will set a messaging action against a component, id and severity
+uvm_set_severity=<component>,<id>,<current_severity>,<new_severity>	This will change the severity of a message against component and id.

Messaging in Sequences

Sequences typically use messaging, either for debug, traceability or to report on the outcome of a built-in checking mechanism.

A sequence object is derived from a uvm_object and therefore does not have the UVM messaging infrastructure built into it like a component object. However, for sequences and sequence_items the sequencer is used as the component from which the message is reported. This changes the way in which the sequence "path" is reported in the UVM messaging. For a sequence, the message path changes to the UVM hierarchical component path to the sequencer on which it is running, followed by a double at sign (@@) followed by the name of the sequence:

```
 UVM_INFO ../ef_sfr_test_pkg/sfr_test_seq.svh(12) @ 0:
uvm_test_top.env.agent.sequencer@@actual_seq [SFR_TEST_SEQ_START]
Starting test
```

If you are working with hierarchical sequences and you are interested in seeing where the sequence fits in the sequence hierarchy, then you need to ensure that whenever you call a sequence's start() method, you assign the handle to the parent sequence to the second argument of the call. If you don't do this, then a null handle is passed by default which means that only the leaf name of the sequence is reported. See line 6 in the body method of the sfr_test_upper_seq sequence in the code excerpt below for an example of how to call a sub-sequence start() method with the second, parent, argument assigned to this, the parent sequence.

```
class sfr_test_seq extends uvm_sequence #(sfr_seq_item);

task body;
  sfr_seq_item item = sfr_seq_item::type_id::create("item");

  `uvm_info("SFR_TEST_SEQ_START", "Starting test", UVM_MEDIUM)
  // Sequence action
  `uvm_info("SFR_TEST_SEQ_END", "Test completed", UVM_MEDIUM)

endtask: body
endclass: sfr_test_seq
```

```
0:class sfr_test_upper_seq extends uvm_sequence #(sfr_seq_item);
1:
2:task body;
3:  sfr_test_seq actual_seq =
sfr_test_seq::type_id::create("actual_seq");
4:
5:  // Using the second argument ensures that the parent sequence
appears in the sequence info messages
6:  actual_seq.start(m_sequencer, this);
7:endtask
8:endclass

// From the test:
task sfr_test::run_phase(uvm_phase phase);
  sfr_test_upper_seq seq =
sfr_test_upper_seq::type_id::create("test_seq");

  phase.raise_objection(this);
  seq.start(env.agent.sequencer);
  phase.drop_objection(this);

endtask: run_phase
```

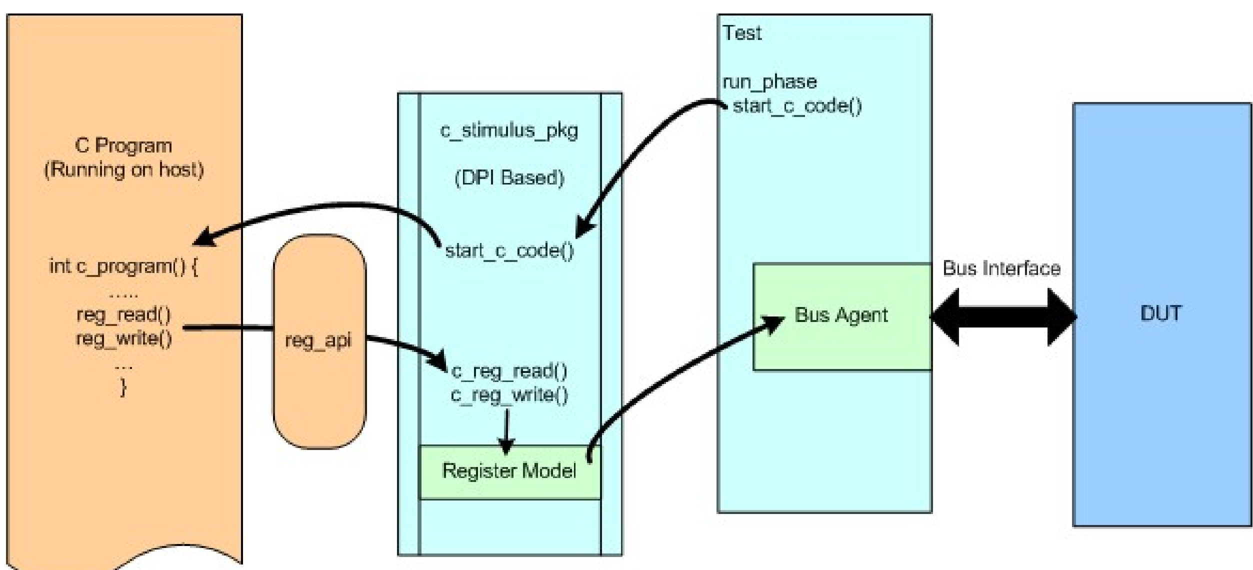
If you follow this approach, then the resultant UVM_INFO message changes to this:

```
 UVM_INFO ../ef_sfr_test_pkg/sfr_test_seq.svh(12) @ 0:
uvm_test_top.env.agent.sequencer@@test_seq.actual_seq
[SFR_TEST_SEQ_START] Starting test
```

Other Stimulus Techniques

CBasedStimulus

Many hardware blocks are designed to interact with software using memory mapped registers. In the final implementation, the system level software, running on a CPU, reads and writes these registers via a bus interface on the hardware block. With UVM sequence based stimulus, accesses to these registers are made via a bus agent, sometimes in a directed way that emulates software accesses, sometimes using constrained random stimulus.



c_stimulus_pkg – Block Diagram

The UVM register model is often used to raise the abstraction of the stimulus generated by these sequences.

However, there is often a requirement to develop c based test stimulus, the reasons for this include:

- A desire to develop device driver code early
- A requirement to have some directed tests that can be run at higher levels of integration, or potentially on a target device
- Additional software engineer resources are available to write tests

One way in which the c stimulus can be applied to the DUT is to insert a CPU or CPU model into a version of the testbench and then compile and execute the c as a program running on the CPU. There is often a significant overhead involved with setting this additional testbench up, and then with simulating the CPU. This article describes a lighter weight alternative that can be used without having to change structure of an existing UVM testbench that contains one or more bus agents. The approach used is to add a C register read/write API for use by C source code, which calls tasks in a SystemVerilog package via the SystemVerilog DPI mechanism to enable the C to make register accesses via the UVM testbench bus agents. The API enables c code to be compiled and then run on the host workstation during the simulation of a UVM environment. The package is called c_stimulus_pkg and comprises a light-weight C-API and two SystemVerilog packages.

Comparison with UVM-Connect

The C Stimulus package is not the same as the UVM-Connect [1] package.

The UVM Connect package encapsulates two main areas of functionality:

- TLM communication between UVM testbenches and SystemC via

1. TLM 1 ports and exports
2. TLM 1 analysis ports and exports
3. TLM 2 sockets

- Providing a means for SystemC to call UVM functions

The primary purpose of the UVM-Connect package is to allow the user to mix SystemC and SystemVerilog components and stimulus. Although UVM-Connect is a very powerful solution, it does not provide a route to creating c or c++ programs that can access hardware registers.

The purpose of the C Stimulus package is to enable c-routines that communicate with hardware registers to access those registers in a DUT hooked up to a UVM bus agent within a UVM verification environment.

C Stimulus Package Overview

Pre-Requisites:

The C Stimulus package assumes the use of a register model in a UVM testbench. That register model should be integrated so that all the possible register accesses can be made via target bus agents.

If a register model is not available, then you will have to write one and integrate it. The process for doing this is described in the register article.

Theory Of Operation:

The C-Stimulus package uses the UVM register model to make accesses to DUT hardware registers via a thin DPI layer. On the software side, a c program makes a hardware register access using an address and a data argument, this access is converted to a UVM register read() or write() call by the c_stimulus_pkg.

The c stimulus is written as normal c, `including the reg_api.h header file which is supplied as part of the c_stimulus_pkg. The UVM test is responsible for starting the c stimulus.

UVM Side Of The c_stimulus_pkg

In order to use the package, the UVM testbench needs to assign a valid handle to the register model before starting the c stimulus at the beginning of the run_phase. A function call is provided in the package to make this easier for the user:

```
//
// function: set_c_stimulus_register_block
//
// Sets the register model handle to the UVM environment register
// model so that c based register accesses can use the register model
//
function void set_c_stimulus_register_block(uvm_reg_block rm);
```

The package contains three tasks which are exported via the SystemVerilog DPI so that they are available to the c-side reg_api layer:

- c_reg_read()

- c_reg_write()
- wait_1ns() - Hardware delay - Wait for n * 1 ns

These tasks are intended to be only used by the reg_api layer.

```
//
// task: c_reg_read
//
// Reads data from register at address
//
task automatic c_reg_read(input int address, output int data);

//
// task: c_reg_write
//
// Writes data to register at address
//
task automatic c_reg_write(input int address, input int data);

//
// task: wait_1n
//
// Wait for n * 1ns
//
task wait_1ns(int n = 1);
```

When either of the read or write methods is called from C code, they go through the following process:

- Get the handle for the register to be accessed via a lookup in the register model using the get_register_from_address() method
- Call a reg.read() or reg.write() method using the register handle
- In the case of a read, return the read data

Pseudo code for the read case is shown in the code snippet below:

```
//
// function: get_register_from_address
//
// Uses the register model to make an lookup of the register
// associated with the address passed to the function.
//
// Returns a handle to the addressed register
//
function uvm_reg get_register_from_address(int address);
  uvm_reg_map reg_maps[$];
  uvm_reg found_reg;

  if(register_model == null) begin
    `uvm_error("c_reg_read", "Register model not mapped for the
c_stimulus package")
  end
```

```
  register_model.get_maps(reg_maps);
  foreach(reg_maps[i]) begin
    found_reg = reg_maps[i].get_reg_by_offset(address);
    if(found_reg != null) begin
      break;
    end
  end

  return found_reg;

endfunction: get_register_from_address

task automatic c_reg_read(input int address, output int data);
  uvm_reg_data_t reg_data;
  uvm_status_e status;
  uvm_reg read_reg;

  read_reg = get_register_from_address(address);
  if(read_reg == null) begin
    `uvm_error("c_reg_read", $sformatf("Register not found at address:
%0h", address))
    data = 0;
    return;
  end
  read_reg.read(status, reg_data);

  data = reg_data;

endtask: c_reg_read
```

The c_stimulus_pkg C API

The C API for the c_stimulus_pkg is defined in a header file called the reg_api.h. It is very light-weight and contains only 4 functions:

```
// reg_api.h
//
//
// function: reg_read
//
// Returns data from register address
//
int reg_read(int address);

//
// function: reg_write
//
// Writes data to register address
```

```
//
void reg_write(int address, int data);

//
// function: register_thread
//
// Called to register a non-default c thread with
// the c_stimulus_pkg context
//
void register_thread();

//
// function: hw_wait_1ns
//
// Hardware delay in terms of 1ns increments
//
void hw_wait_1ns(int n);
```

Starting The C Code

The read and write functions have to be called from the C program. In order to start the C program, a DPI context task needs to be called from SystemVerilog and a default for this is provided within the C Stimulus package - start_c_code(). The C code needs to implement a function call with the same name, either with the c program or a call to a function that contains the c program. Note that the normal name of a c program, "main", should not be used. The UVM testbench should call the start_c_code() function during an active UVM phase such as the run_phase in order to start C execution.

Provision For Multiple Bus Targets

Most block level testbenches only deal with a single bus interface, but with more complex DUTs, there may be several bus interfaces which are used to access the registers. Provided the different bus masters are in the register model's register map and the UVM testbench supports register based accesses to all the DUT interfaces, then the person writing the c code only has to worry about reading and writing from the registers at the correct addresses.

Multiple C Threads

In some circumstances, there may be a requirement to run multiple C threads and this can easily be accommodated by the user making calls to the additional C program threads via his own DPI imports in his own package. The test package is usually the most convenient place to do this.

Note that:

- The DPI imports must be "context" imports
- If a DPI import is defined, then there must be a matching c function declared, otherwise there will be at least an elaboration warning

```
// Example of importing a c test routine via the DPI
import "DPI-C" context task a_c_test_routine;
```

On the c side a register_thread() method should be called at the beginning of a c-thread. This is used to register the DPI context of the c-thread as the c_stimulus_pkg.

```
// In the UVM test:
//
task run_phase(uvm_phase phase);
  phase.raise_objection(this);

  fork
    start_c_code(); // Default c thread
    a_c_function(); // Additional c thread
    // HW Stimulus:
    v_seq.start(null);
  join

  phase.drop_objection(this);

endtask: run_phase

//
// On the c-side:
//

int a_c_test_routine() {
  // Declare variables

  register_thread(); // Must be called

  // Rest of thread code

  return 0;
}
```

More Than One C Based Test

The default package assumes that only one c based test will be written and that the c-side will be started using start_c_code(). However, if more than one c based test is required, then one of three strategies can be used:

- Recompile the c code before running each testcase in order to make sure that start_c_code() calls the correct c-side function.
- Import a new c function for the c-side of each new testcase in the test package - using the same approach as outlined in the section describing multiple threads.
- Compile each c code into a separate shared object using the advanced DPI compilation flow and only load that shared object.

The recommended approach is to add a new c function to wrap the thread for a new c based test and to add a DPI import for the c side function into the test package.

Handling Interrupts

In the "real world" a hardware interrupt causes a CPU execution thread to suspend, and then jump to some interrupt handler code to service the interrupt and then return to the execution thread once the handler routine has completed.

In order to simulate the effect of a hardware interrupt, an additional package should be used - isr_pkg. This provides a task called interupt_service_routine, this raises a flag that blocks other c threads from accessing the hardware registers until the interrupt_service_routine completes.

On the c side, a function called start_isr() is used to wrap the interrupt service routine. This function should **NOT** call the register_thread() method.

This is an approximation to what would happen with a real CPU, since the main c thread will continue to execute until it blocks on a register access.

Package Download:

The C Stimulus package can be downloaded here:

An Example Of Using The C Stimulus Package

The following example is an adaptation of the SPI testbench used as one of the cookbook register examples. The example shows how to use a c routine to test the DUT, and how to use a c interrupt service routine to handle interrupts. It uses the same testbenchfiles as the original example, but the test package adds imports of the C Stimulus and the ISR packages and a new test class which starts the c routine during its run phase.

UVM Use Model

In order to use the c interface, a UVM test has to assign the register model handle in the c_stimulus_pkg package to the testbench register model and then call the c thread(s) to execute, this is acheived using the set_c_stimulus_register_block() method.

If there is only one c thread to execute, then the default call to the c thread is start_c_code() - this should be matched by a function in the c application of the same name. If multiple c threads are used, then a call to each of these should be declared in a package (possibly the test package) as DPI imports.

As an example, the run_phase for a simple example test would be:

```
// From inside a test `included into a package containing the following
 imports:
import c_stimulus_pkg::*;
import isr_pkg::*;


// This task starts the c program that then calls back into
// the UVM simulation
//
// It also monitors the interrupt line from the SPI block
// and calls the interrupt service routine when it is asserted
//
task spi_c_int_test::run_phase(uvm_phase phase);
  spi_tfer_seq spi_seq = spi_tfer_seq::type_id::create("spi_seq");

  phase.raise_objection(this, "Test Started");
```

```
  `uvm_info("run_phase", "starting c code", UVM_LOW)

  set_c_stimulus_register_block(spi_rm); // Assign the register model
handle

  fork
    start_c_code(); // Start the c-side test routine
    // Respond to SPI transfers:
    begin
      forever begin
        spi_seq.BITS = 0;
        spi_seq.rx_edge = 0;
        spi_seq.start(m_env.m_spi_agent.m_sequencer);
        spi_rm.ctrl_reg.char_len.get(spi_seq.BITS);
        spi_rm.ctrl_reg.rx_neg.get(spi_seq.rx_edge);
        spi_seq.start(m_env.m_spi_agent.m_sequencer);
      end
    end
    begin
      forever begin
        m_env_cfg.wait_for_interrupt();
        interrupt_service_routine(); // Start the c-side interrupt
service routine
      end
    end
  join_any
  `uvm_info("run_phase", "c code finished", UVM_LOW)
  phase.drop_objection(this, "Test Finished");

endtask: run_phase
```

This test also illustrates the use of an interrupt. In the run_phase() method, an interrupt line is monitored via a configuration monitoring method (see the article on signal_wait for more details). When an interrupt is detected, the interrupt service routine is called.

Software Use Model

The c code used in this example #includes the reg_api.h file, so that it can call the UVM register access methods. The c test code and the interrupt service routine are wrapped by the start_c_code() and start_isr() functions which are the default DPI imports supported by the c_stimulus_pkg and the isr_pkg.

```
#include "spi_regs.h" // Defines for register offsets etc
#include "reg_api.h"  // UVM C stimulus register layer API

int int_flag = 0;

void spi_int_test() {
  int no_chars = 1;
  int format = 0;
```

```
    int divisor = 2;
    int slave_select = 1; int control
    = 0;
    int i = 0;
    int   data_0   =   0x12345678;   int
    data_1 = 0x87654321; int data_2 =
    0x90901212;     int     data_3     =
    0x5a6b7c8d; int status;
    int data;

    reg_write(DIVIDER, divisor);

    while(i < 10) {
        int_flag = 0;
        control = no_chars + (format << 9) + 0x3000; reg_write(CTRL,
        control);
        reg_write(SS, slave_select);
        reg_write(TX0, data_0); reg_write(TX1,
        data_1); control = control + 0x100;
        reg_write(CTRL, control); while(int_flag
        == 0) {
            status = reg_read(CTRL); data =
            reg_read(SS);
        }
        no_chars = no_chars++; format
        = format++; if(format == 8) {
            format = 0;
        }
        slave_select = slave_select << 1;
        if(slave_select = 0x100) { slave_select
            = 1;
        } i++;
    }

}

void spi_isr() { int
    status; int rx_data0;
    int rx_data1; int
    rx_data2; int
    rx_data3;
```

```
  status = reg_read(CTRL);
  reg_write(SS, 0x0);
  rx_data0 = reg_read(RX0);
  rx_data1 = reg_read(RX1);
  rx_data2 = reg_read(RX2);
  rx_data3 = reg_read(RX3);
  int_flag = 1;
}


int start_c_code () {
  spi_int_test();
  return 0;
}

int start_isr () {
  spi_isr();
  return 0;
}
```

Extending The Example To Run Another C Stimulus Test

The same testbench can be extended to run with another c test routine by adding a DPI import for the c test routine to the test package, and by adding another test class to the package:

```
// In the spi_test_lib_pkg:

// DPI Imports for c based routines:
import "DPI-C" context task spi_c_poll_test_routine();

// C based tests:
`include "spi_c_int_test.svh"
`include "spi_c_poll_test.svh"

// The run method of spi_c_poll_test
//
// This task starts the c program that then calls back into
// the UVM simulation
//
task spi_c_poll_test::run_phase(uvm_phase phase);
  spi_tfer_seq spi_seq = spi_tfer_seq::type_id::create("spi_seq");
  uvm_reg_data_t reg_data;


  phase.raise_objection(this, "Test Started");
  `uvm_info("run_phase", "starting c code", UVM_LOW)

  set_c_stimulus_register_block(spi_rm);
```

```
  fork
    spi_c_poll_test_routine(); // Calling the imported C routine
    // Respond to SPI transfers:
    begin
      forever begin
        spi_seq.BITS = 0;
        spi_seq.rx_edge = 0;
        spi_seq.start(m_env.m_spi_agent.m_sequencer);
        spi_seq.BITS = spi_rm.ctrl_reg.char_len.get();
        spi_seq.rx_edge = spi_rm.ctrl_reg.rx_neg.get();
        spi_seq.start(m_env.m_spi_agent.m_sequencer);
      end
    end
  join_any
  `uvm_info("run_phase", "c code finished", UVM_LOW)
  phase.drop_objection(this, "Test Finished");

endtask: run_phase
```

The corresponding c test routine needs to make the register_thread() API call at the beginning:

```
#include "spi_regs.h" // Defines for register offsets etc
#include "reg_api.h"  // DPI Register Hardware access layer API


int spi_c_poll_test_routine() {
  int no_chars = 1;
  int format = 0;
  int divisor = 2;
  int slave_select = 1;
  int control = 0;
  int i = 0;
  int data_0 = 0x12345678;
  int data_1 = 0x87654321;
  int data_2 = 0x90901212;
  int data_3 = 0x5a6b7c8d;
  int status;
  int data;

  register_thread(); // To register this thread with the c_stimulus_pkg
 DPI context

  reg_write(DIVIDER, divisor);

  //
  // etc
  //
```

```
  return 0;
}
```

Compilation and Simulation Process

The compilation process for the Package and the c code, using Questa vlog compiler, is as follows:

- Compile the c_stimulus_pkg.sv file, and if required, the isr_pkg.sv, generating a DPI header file
- Compile the test package for any UVM tests that are relying on c-side threads other than the default, generating a DPI header
- Compile the reg_api.c file
- Compile the application c code

When the simulation is invoked, then Questa will automatically create the necessary shared object used in simulation.

```
To generate the DPI header file for the package:

vlog $(C_STIMULUS_PKG_HOME)/c_stimulus_pkg.sv -dpiheader sv_dpi.h
vlog $(C_STIMULUS_PKG_HOME)/isr_pkg.sv -dpiheader sv_dpi.h

If there is a non-default c test routine in a test package:
vlog +incdir+$(TEST_PKG_HOME) $(TEST_PKG_HOME)/test_pkg.sv -dpiheader
sv.dpi.h

To compile the reg_api:
vlog +incdir+$(C_STIMULUS_PKG_HOME) $(C_STIMULUS_PKG_HOME)/reg_api.c

To compile the c thread code:
vlog +incdir+$(C_CODE_HOME) $(C_CODE_HOME)/my_c_code.c -ccflags
-I$(C_STIMULUS_PKG_HOME)

To run the simulation loading the package:
vsim top_tb +UVM_TESTNAME=spi_c_int_test
```

Caveats

The c side of the API is implemented as function calls where the address of the register is passed as an argument. The easiest way to abstract these addresses is to use #defines in a header file.

There are alternative methods of implementing a register interface API which involve using an array of structs to map the registers into memory space, register accesses then take place by using pointers to these structs. Unfortunately, this approach cannot be used with the API provided. In order to be able to support this style of interface it would be necessary to either add a compiler option or implement a memory management unit that would throw an exeception which would allow the access to be taken care of using the simple API provided. Both of these options are outside the scope of the solution provided.

Example Download

To download the example that illustrates:

1. The use of c stimulus
2. Interrupt handling using c stimulus
3. The use of an additional c based thread for another test case

References

[1] http://verificationacademy.com/verification-methodology/uvm-connect

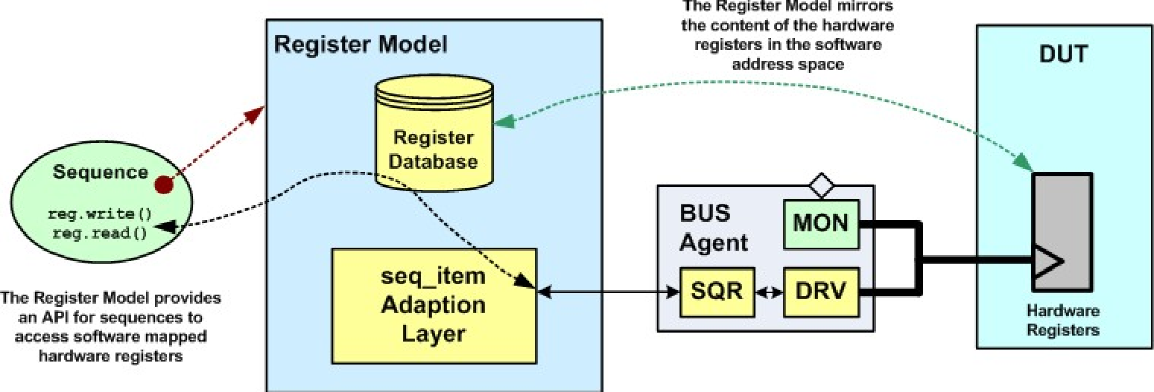
Register Abstraction Layer

The UVM Register Package

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.



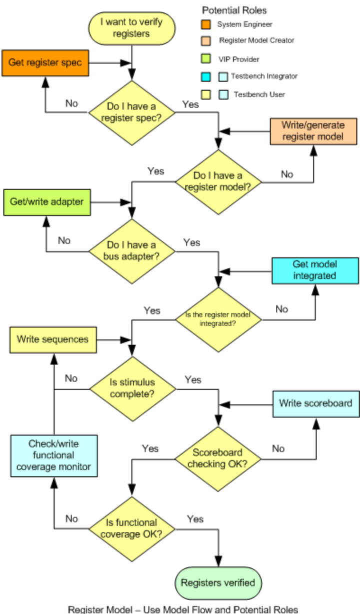
UVM Register Model Functional Overview

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 faciliate 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.



Register Model – Use Model Flow and Potential Roles

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 Unecessary

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	ComplexAddressMaps	Advanced register model addressing considerations	Background	Essential
5	QuirkyRegisters	Implementing 'Quirky' registers	Essential	Essential
6	ModelCoverage	Adding coverage models	Background	Essential
7	BackdoorAccess	Using and specifying back door accesses	Background	Essential
8	Generation	Generating a register model	Essential	Unecessary

Integrating A Register Model

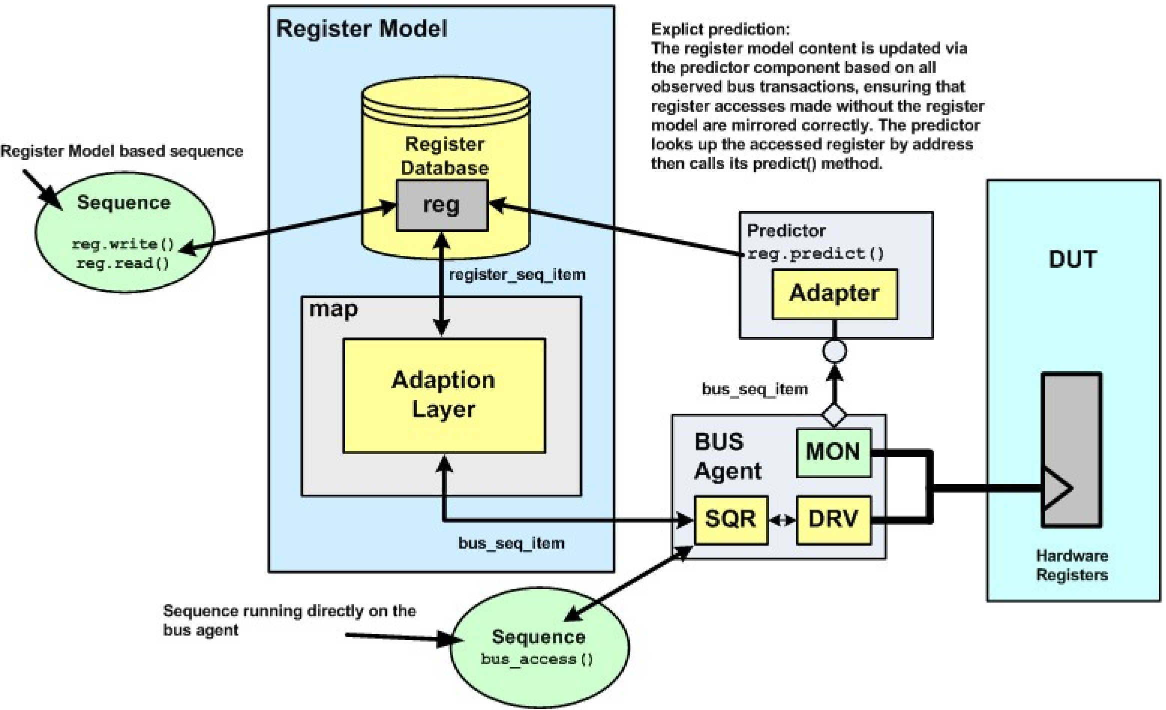
Integration Pre-requisites

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 enviornment 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.



Explicit Prediction Use 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 reuseable 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 reuseable 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 mantain 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	
Memory Sub-System Testbench	

Register Model Overview

Introduction

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 modeled 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 modeled 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 modeled 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.

Register Model Layer Overview

The UVM register model is built up in layers:

Layer	Register base class	Description	Diagram
Fields	*uvm_reg_field*	Bit(s) grouped according to function within a register	F 0 1 2
Register	*uvm_reg*	Collection of fields at different bit offset	R 0:2 F1 4:5 F2 7:11 F3
Memory	*uvm_mem*	Represents a block of memory which extends over a specified range	M x300

Block	*uvm_block*	Collection of registers (Hardware block level), or sub-blocks (Sub-system level) with one or more maps. May also include memories.	
Map	*uvm_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 Model Data Types

The register model uses some specific data types in order to standardize 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.

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 modeled 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 modeled 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.

reg map

Offset	Reg/Mem/Block
0x70	Register_A
0x84	Register_B
0xFF00_0000	Memory
0xC000_F000	Sub-block map

Target Agent Adapter Handle

Target Sequencer Handle

Register Map Contents

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 organized 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.

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 finalize the address mapping and to ensure that the model cannot be altered by another user.

Hierarchical Register Blocks

The register block in the previous 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
```

Register Model Structure

Introduction

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 modeled 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 modeled 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 modeled 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.

Register Model Layer Overview

The UVM register model is built up in layers:

Layer	Register base class	Description	Diagram
Fields	*uvm_reg_field*	Bit(s) grouped according to function within a register	
Register	*uvm_reg*	Collection of fields at different bit offset	
Memory	*uvm_mem*	Represents a block of memory which extends over a specified range	
Block	*uvm_block*	Collection of registers (Hardware block level), or sub-blocks (Sub-system level) with one or more maps. May also include memories.	
Map	*uvm_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 Model Data Types

The register model uses some specific data types in order to standardize 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.

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 modeled 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 modeled 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.

reg map

Offset	Reg/Mem/Block
0x70	Register_A
0x84	Register_B
0xFF00_0000	Memory
0xC000_F000	Sub-block map

Target Agent Adapter Handle

Target Sequencer Handle

Register Map Contents

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 organized 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.

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 finalize the address mapping and to ensure that the model cannot be altered by another user.

Hierarchical Register Blocks

The register block in the previous 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
```

Complex Address Maps

In SoC design, the address mapping of registers and memory is often more complex than a single map. When there are several masters in a system, then they often have a different view of the register map, i.e. the same register will appear at different addresses depending on which bus master is accessing it. Another common scenario is for the address map to be dynamic.

Multiple Address Mapping

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
```

Dynamic Address Mapping

In SoC designs, address maps can be changed on the fly for reasons including the following:

- Security - Registers are made inaccessible in different states
- Reconfiguration - After discovery or re-enabling of a hardware resource, the address map has to be reconfigured
- An additional host is added to the system, requiring a new address view to be added to the address map
- Virtualisation - Where during an interrupt different virtual machines have to take over common resources

The UVM register map supports dynamic re-mapping by allowing the user to unlock and reconfigure a register block. After the unlock_model() operation, the unregister() function can be used to remove a register map from the block, additional register maps can be added to the block, and map register content can be unregistered or added to.

Since a SoC register model may have to support several different register map configurations that could change dynamically during a test, the SoC register block should contain re-map functions to make the operation easy to use.

Dynamically Removing/Re-Adding Registers and Memories

Assuming that we are using the mem_ss_reg_block defined previously, there might be a chip level security mode where certain registers or memory regions would not be available to users. The following example shows an example method for going into a secure mode by unmapping mem3 and its associated registers, and then another for exiting the secure mode by re-instating the mapping of mem3 and its registers.

```
//
// 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

// Removes mem3 and the registers associated with it from the AHB_map
function void security_demap();
  unlock_model();
  AHB_map.unregister(mem3_offset);
  AHB_map.unregister(mem3_range);
```

```
  AHB_map.unregister(mem3);
  lock_model();
endfunction

// Re-instates mem3 and the registers associated with it into the
AHB_map
function void security_remap();
  unlock_model();
  AHB_map.add_reg(mem_3_offset, 32'h00000010, "RW");
  AHB_map.add_reg(mem_3_range, 32'h00000014, "RW");
  AHB_map.add_mem(mem_3, 32'h0001_0000, "RW");
  lock_model();
endfunction

endclass: mem_ss_reg_block
```

Dynamically Changing An Address Map

The following code example illustrates a method for changing an address map to a new mapping, and another method to re-mapped it back to its original configuration. Again, it is based on adding methods to the mem_ss_reg_block.

```
//
// Memory sub-system (mem_ss) register & memory block
//
class mem_ss_reg_block extends uvm_reg_block;

// Remaps the contents of the AHB_map for host_B
function void remap_for_host_B();
  unlock_model();
  unregister(AHB_map); // Unregisters the address map
  AHB_map = null;      // Delete the original AHB_map
  // Create and configure the new version of the AHB_map
  AHB_map = create_map("AHB_map", 'h0, 4, UVM_LITTLE_ENDIAN, 1);
  // Add registers and memories to the AHB_map
  AHB_map.add_reg(mem_1_offset, 32'h20000000, "RW");
  AHB_map.add_reg(mem_1_range, 32'h20000004, "RW");
  AHB_map.add_reg(mem_2_offset, 32'h20000008, "RW");
  AHB_map.add_reg(mem_2_range, 32'h2000000c, "RW");
  AHB_map.add_reg(mem_3_offset, 32'h20000010, "RW");
  AHB_map.add_reg(mem_3_range, 32'h20000014, "RW");
  AHB_map.add_reg(mem_status, 32'h20000018, "RO");
  AHB_map.add_mem(mem_1, 32'hE000_0000, "RW");
  AHB_map.add_mem(mem_2, 32'hC000_0000, "RW");
  AHB_map.add_mem(mem_3, 32'h4000_0000, "RW");
  lock_model();
endfunction

// Re-instates the default AHB_map for host_A
```

```
function void remap_to_default();
  unlock_model();
  unregister(AHB_map);
  AHB_map = null;
  AHB_map = create_map("AHB_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");
  lock_model();
endfunction

endclass: mem_ss_reg_block
```

Specifying Registers

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 organized 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

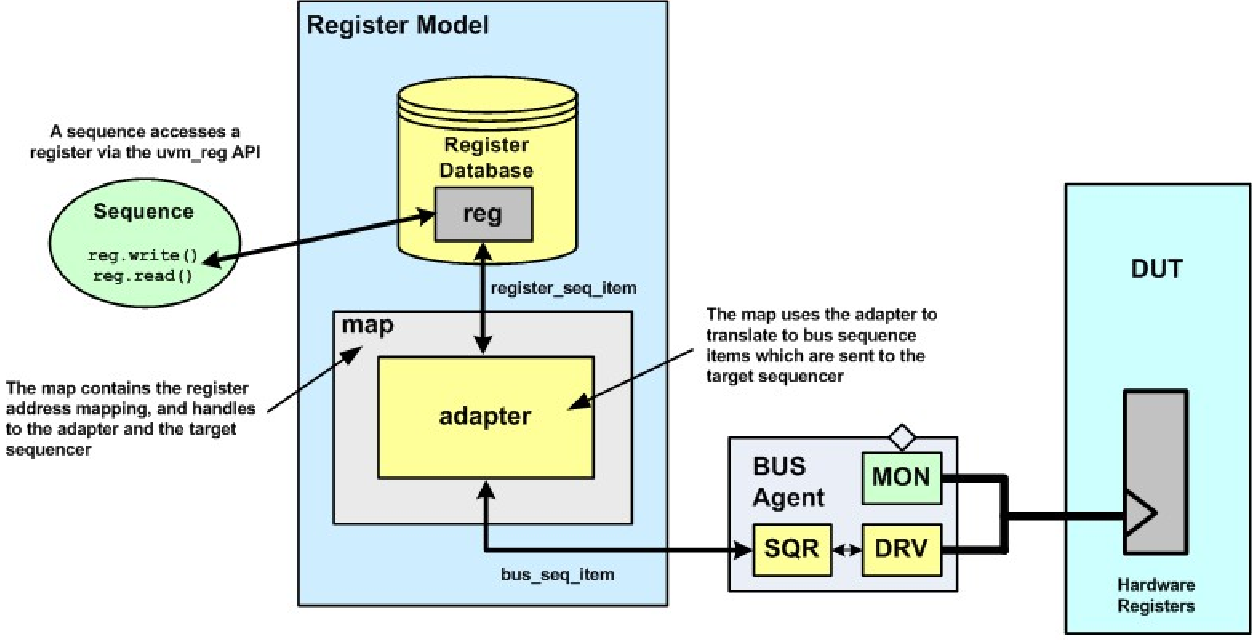
Note: In the SPI master the RXn and TXn registers actually map onto the same hardware flip-flops and during character transmission and reception their content changes. Therefore from the software interface abstraction the TX registers are write only, and the RX registers are read only, and the Rx content 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.

The Register Layer 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 register sequence items. The adapter should be implemented by extending the uvm_reg_adapter base class.



The Register Adapter

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	uvm_reg_addr_t	Address field, defaults to 64 bits
data	uvm_reg_data_t	Read or write data, defaults to 64 bits
kind	uvm_access_e	UVM_READ or UVM_WRITE
n_bits	unsigned int	Number of bits being transferred
byte_en	uvm_reg_byte_en_t	Byte enable
status	uvm_status_e	UVM_IS_OK, UVM_IS_X, UVM_NOT_OK

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.

uvm_reg_adapter	
Methods	**Description**
reg2bus	Overload to convert generic register access items to target bus agent sequence items
bus2reg	Overload to convert target bus sequence items to register model items
Properties (Of type bit)	**Description**
supports_byte_enable	Set to 1 if the target bus and the target bus agent supports byte enables, else set to 0
provides_responses	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:

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.

Integrating a UVM Register Model in a TestBench - Overview

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.

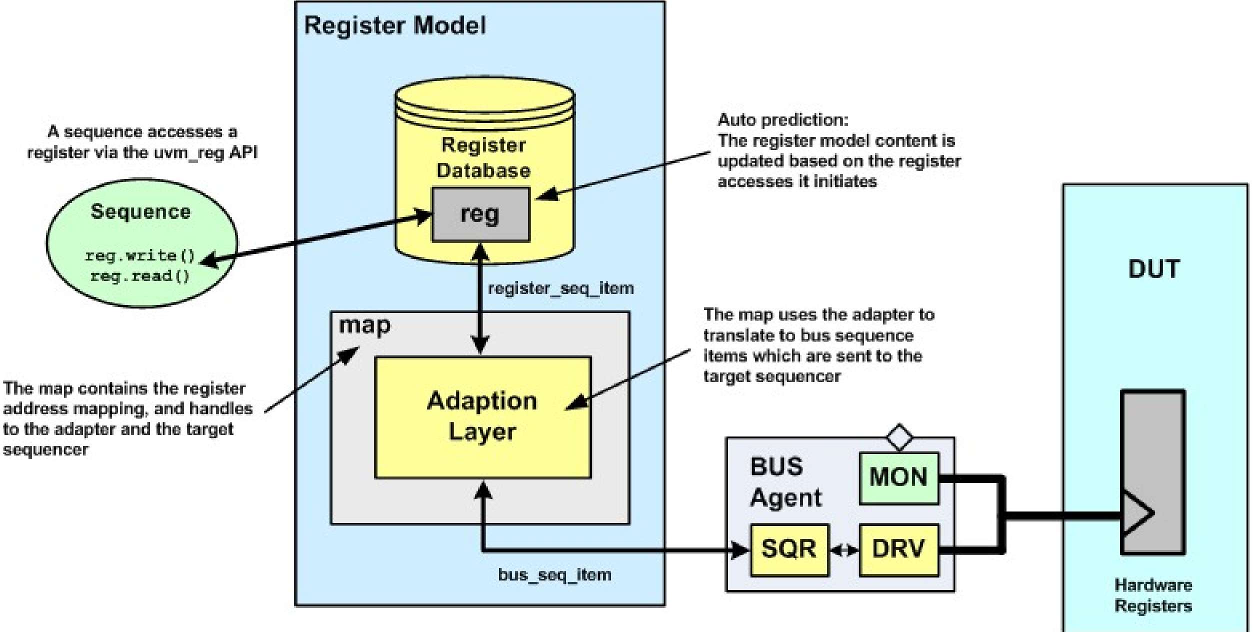
In order to make back door accesses to the DUT, the register model uses hdl paths which are used by simulator runtime database access routines to peek and poke the hardware signals corresponding to the register. The register model is updated automatically at the end of each back door access cycle. The 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 one 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.



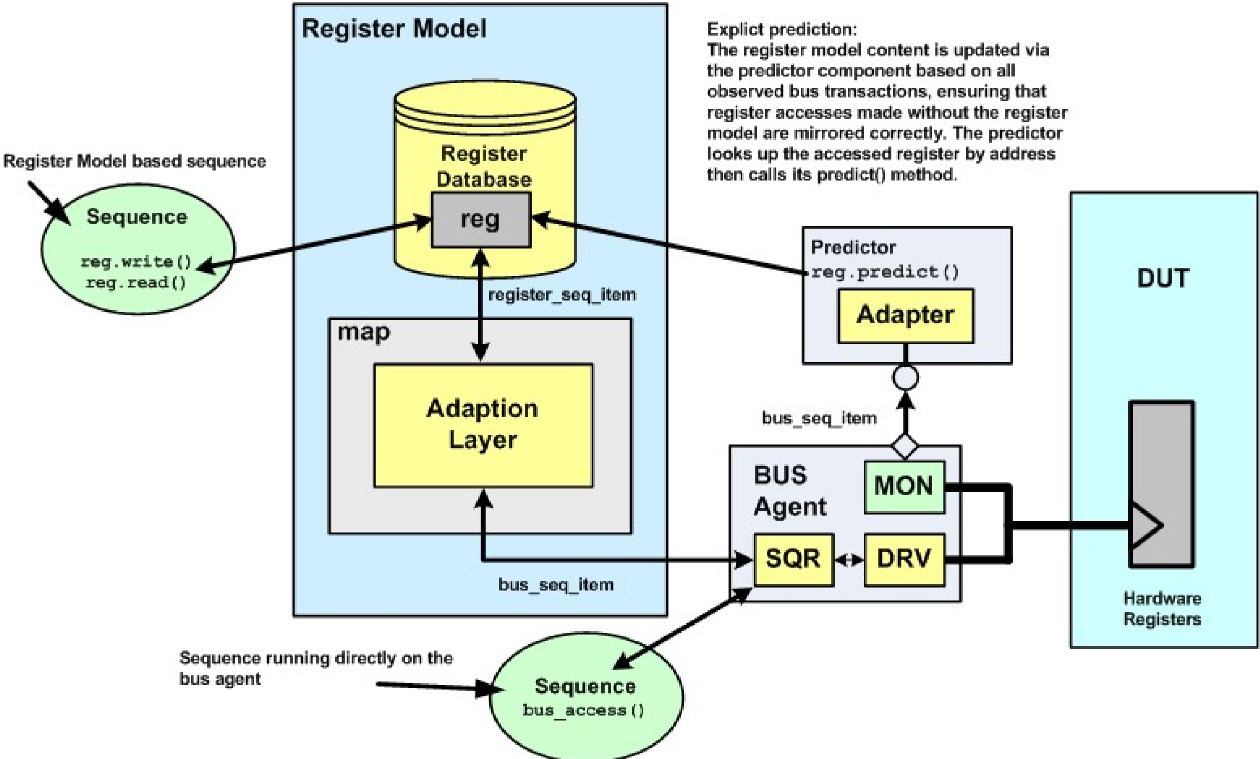
Auto Prediction Use Model

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 wil 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.



Explicit Prediction Use Model

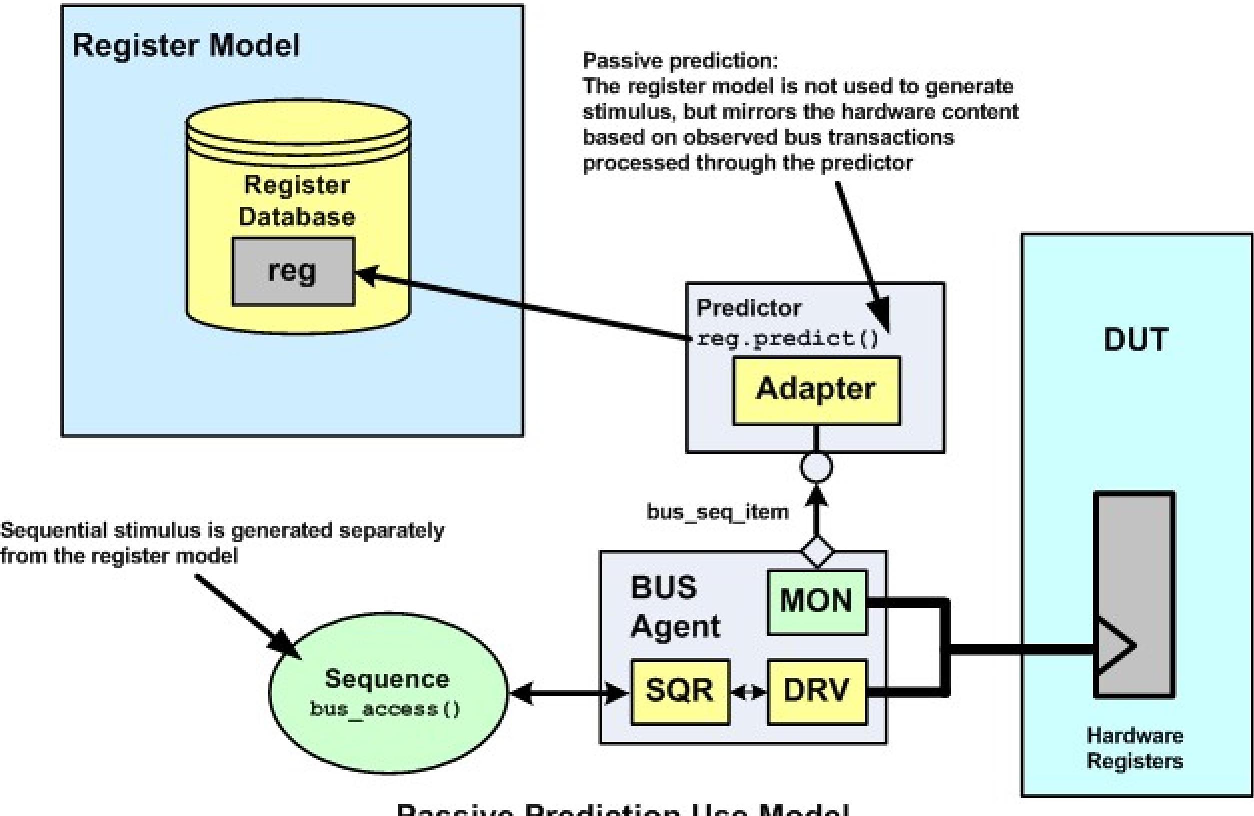
The main advantage of using explicit prediction is that it keeps the register model up to date with all accesses that occur on the target bus interface. The configuration also has more scope for supporting vertical reuse where accesses to the DUT may occur from other bus agents via bus bridges or interconnect 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 place between a monitor and a Predictor.

Passive Prediction

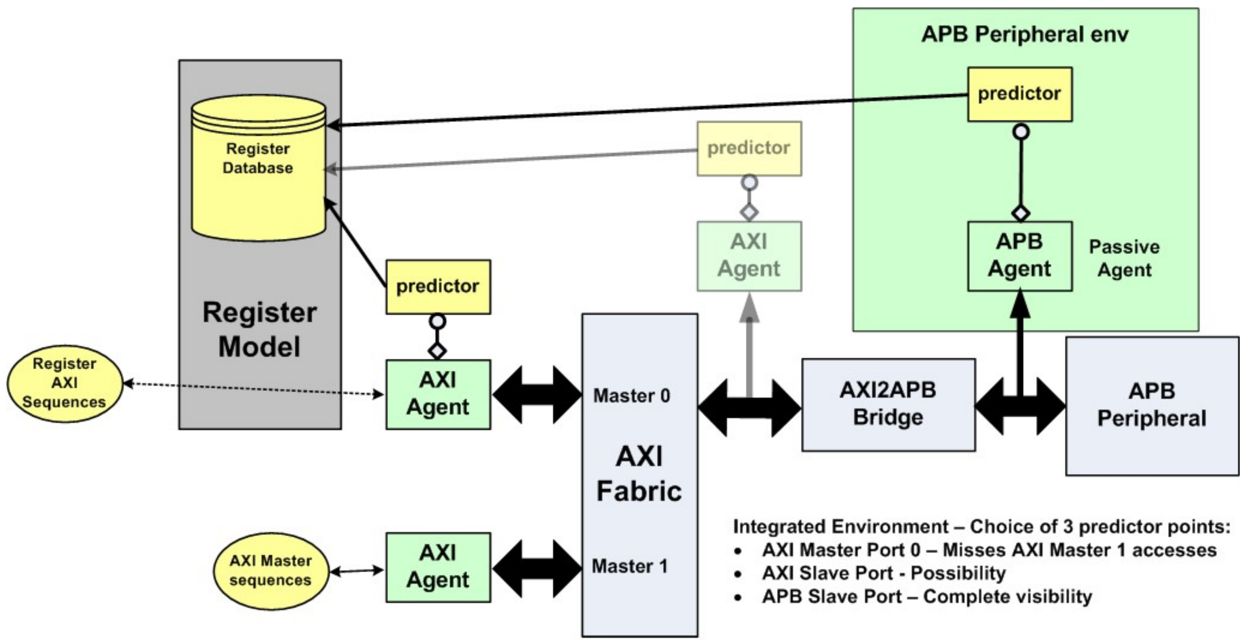
In passive prediction the register model takes no active part in any accesses to the DUT, but it is kept up to date by the predictor when any front door register accesses take place.



Passive Prediction Use Model

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.



Integrated Environment – Explicit Prediction Model

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.

Integrating a UVM Register Model in a TestBench - Implementation

The integration process for the register model involves constructing it and placing handles to it inside the relevent 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** a 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 are not constructed and initialized which means that subsequent testbench code that accesses the register model will fail, most likely with null handle errors.

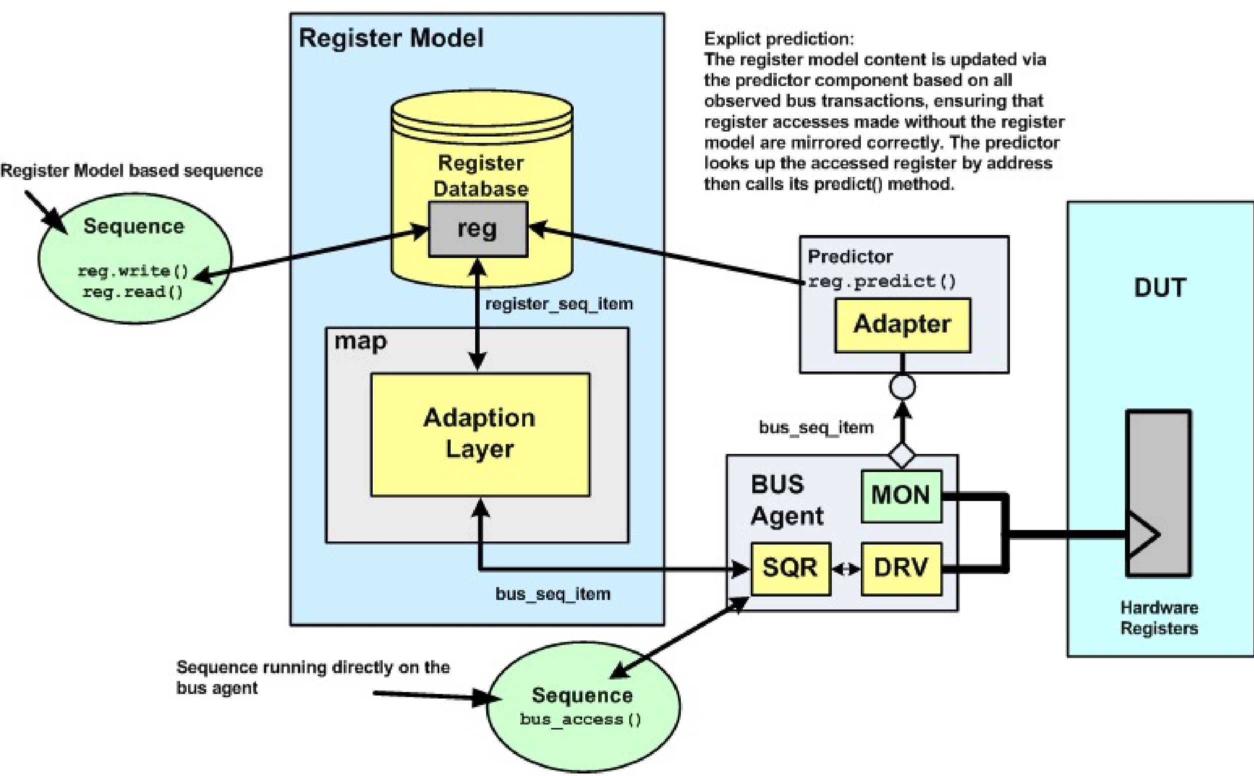
```
//
// From the SPI Test base class
//
// Build the env, create the env configuration
// including any sub configurations and assigning virtual interfaces
function void spi_test_base::build_phase(uvm_phase phase);
  // env configuration
  m_env_cfg = spi_env_config::type_id::create("m_env_cfg");
  // 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.



Explicit Prediction Use Model

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.

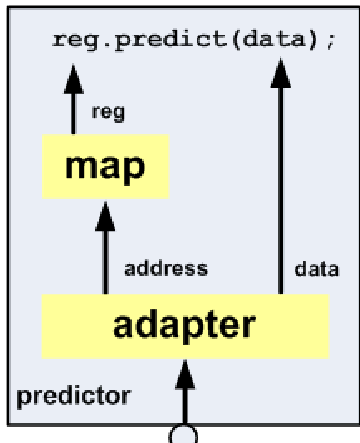
In the following code from the SPI block level env connect method, the APB map from the SPI register block is layered onto the APB agent sequencer using the reg2apb adapter class using the set_sequencer() method in

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
endfunction: connect_phase
```

Register Prediction

By default, the register model uses a process called explicit_prediction 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 parameterized with the type of the target bus analysis transaction. It contains handles for the target register bus adapter and the register model address map that are 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 model's 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 specialization 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
    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:

"Quirky" Registers

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 base class. The register base class can be used to describe the behavior of many different registers - for example clear-on-read (RC), write-one-to-set (W1S), write-zero-to-set (W0S). These built-in behaviors are set using attributes. Setting the attribute causes 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 IEEE 1800.2 LRM has 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 UVM 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.

```
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
```

A Custom Quirky Register

If your quirky register behavior does not match either of the library register models, 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).

```
always @(posedge PCLK) begin
   if(PRESETn == 0) begin
      id_register_pointer <= 0;
      id_register_value <= '{'ha0, 'ha1, 'ha2, 'ha3, 'ha4,
                                          'ha5, 'ha6, 'ha7, 'ha8, 'ha9}; current_value <=
      32'ha0;
   end
   else begin
      if(PSEL & PENABLE) begin
         if(PWRITE) begin
            case(PADDR)
               // A write to the ID register overwrites the register pointer
               `ID : begin
                           if(PWDATA < 10) begin
                              id_register_pointer <= PWDATA[3:0];
                           end
                           else begin
                              id_register_pointer <= 0;
                           end
                        end
               `R_W : W_reg <= PWDATA;
            endcase
         end
         else begin
            // A read from the ID register advances the id_register_pointer
            if(PADDR == `ID)begin if(id_register_pointer == 9)
               begin
                  id_register_pointer <= 0;
               end
               else begin
                  id_register_pointer <= id_register_pointer + 1;
               end
            end
         end
      end
      if(~PENABLE) begin
         current_value <= id_register_value[id_register_pointer];
      end
   end
end

always @(*) begin
   if(PSEL) begin
      case(PADDR)
                  // Return the current_value of the id register
```

```
      `ID: PRDATA = current_value;
      `R_W: PRDATA = R_reg;
      default: PRDATA = 0;
    endcase
  end
  else begin
    PRDATA = 0;
  end
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);
  endfunction: build

  `uvm_object_utils(id_register)

endclass : id_register
```

The id_register builds the ID field in the build() routine, just like any other register.

ID Register Model Field

The ID register field is a simple implementation of the functionality required. The post_read() and post_write() call-back methods allow the register field functionality to be modeled.

Strictly speaking, the ID field is not modeling the behavior of an id register, it is checking the DUT implementation of the register.

```
// The ID Register (field).
// Each successive 'read' operation returns the next item
// from a list. When the end of the list is reached, it wraps
// around to the beginning.
// This list is a0, a1, ...a9. (10 values)
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};

  int current_value;

  function new(string name = "id_register_field");
    super.new(name);
    current_value = id_register_value[0];
  endfunction

  task post_read(uvm_reg_item rw);
    if(value != current_value) begin
      `uvm_error("ID_REG_CHECK", $sformatf("Wrong ID value: id_ptr:%0d
id_val:%0h read_vale:%0h", id_register_pointer, current_value, value))
    end
    id_register_pointer++;
    if (id_register_pointer >= id_register_pointer_max) begin
        id_register_pointer = 0;
    end
    current_value = id_register_value[id_register_pointer];
  endtask

  task post_write(uvm_reg_item rw);
    id_register_pointer = value;
    if (id_register_pointer >= id_register_pointer_max) begin
        id_register_pointer = 0;
    end
    current_value = id_register_value[id_register_pointer];
  endtask


endclass
```

When the post_read() method is called, it first checks that the current_value of the field is the same as the value read back from the DUT. Then it updates the current_value of the field, according to the incremented value of the id_register_pointer, ready for the next register read.

The post_write() method, writes the value into the id_register_pointer and sets up the current_value for the next read cycle.

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.

Register Model Coverage

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 initialized 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 summarized here:

Method	Description
Overall Control	
uvm_reg::include_coverage(uvm_coverage_model_e)	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
Build Control	
build_coverage(uvm_coverage_model_e)	Used to set the local variable m_has_cover to the value stored in the resource database against the "include_coverage" key
has_coverage(uvm_coverage_model_e)	Returns true if the coverage type is enabled in the m_has_cover field
add_coverage(uvm_coverage_model_e)	Allows the coverage type(s) passed in the argument to be added to the m_has_cover field
Sample Control	
set_coverage(uvm_coverage_model_e)	Enables coverage sampling for the coverage type(s), sampling is not enabled by default
get_coverage(uvm_coverage_model_e)	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.

The values of the uvm_coverage_model_e enumerated type are:

uvm_coverage_model_e enum value	Description
UVM_NO_COVERAGE	No coverage enabled
UVM_CVR_REG_BITS	Coverage enabled for register bits
UVM_CVR_ADDR_MAP	Coverage enabled for individual register and memory adddresses
UVM_CVR_FIELD_VALS	Coverage enabled for register fields
UVM_CVR_ALL	All types of register coverage are enabled

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
//
```

Backdoor Accesses

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 bypasses 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
Bypasses 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 ***vopt +acc*** switch. See the Questa user documentation for more information.

```
# Compile my design so that all of it has registers visible:
vlog my_design
vopt my_design +acc -o opt
vsim opt +UVM_TESTNAME=reg_backdoor_test
```

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
```

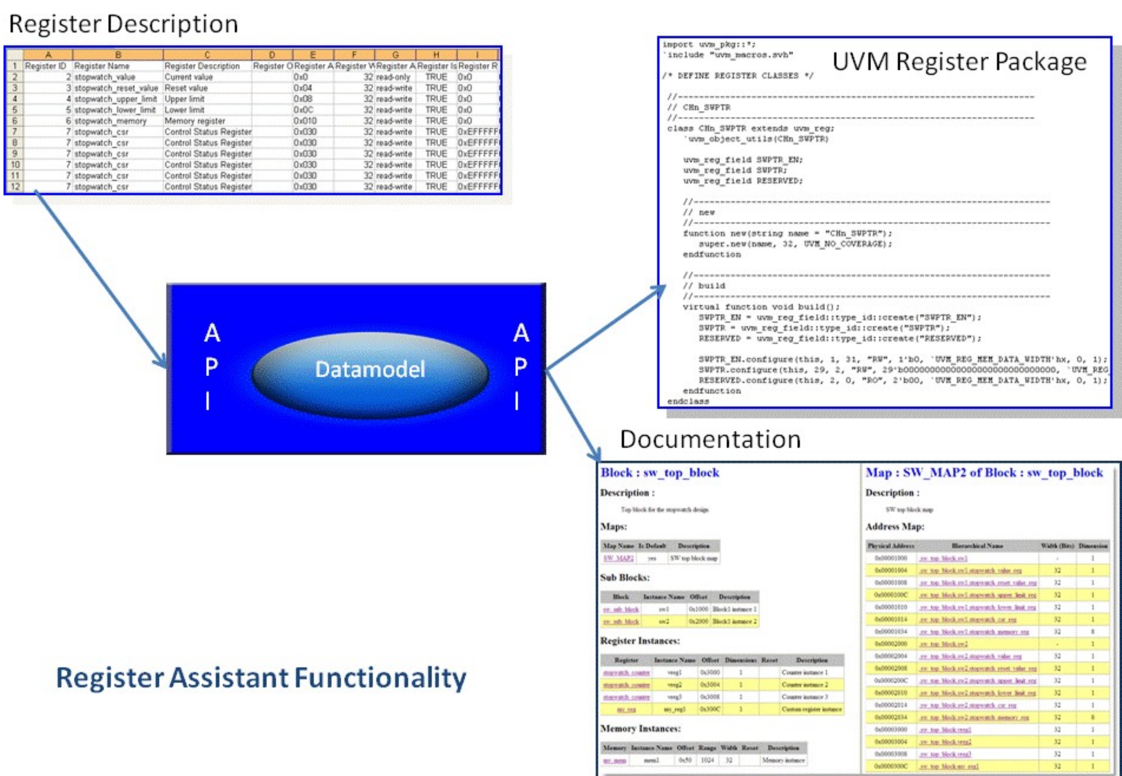
Generating Register Models

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's Register Assistant.

As an option to Mentor's Questa Simulator, 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.



Register-Level Stimulus

Stimulus Abstraction

Stimulus that accesses memory mapped registers 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 programed.

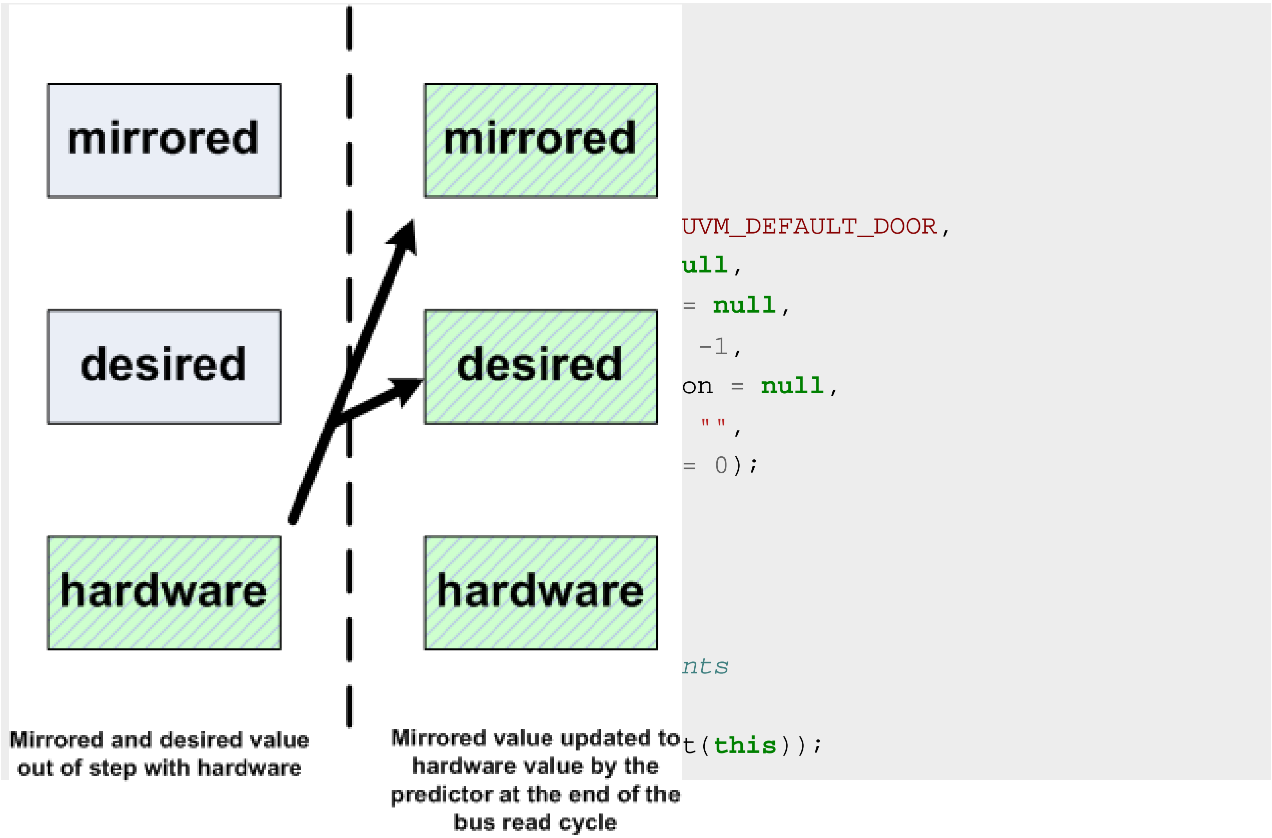
UVM Register Access Methods

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.

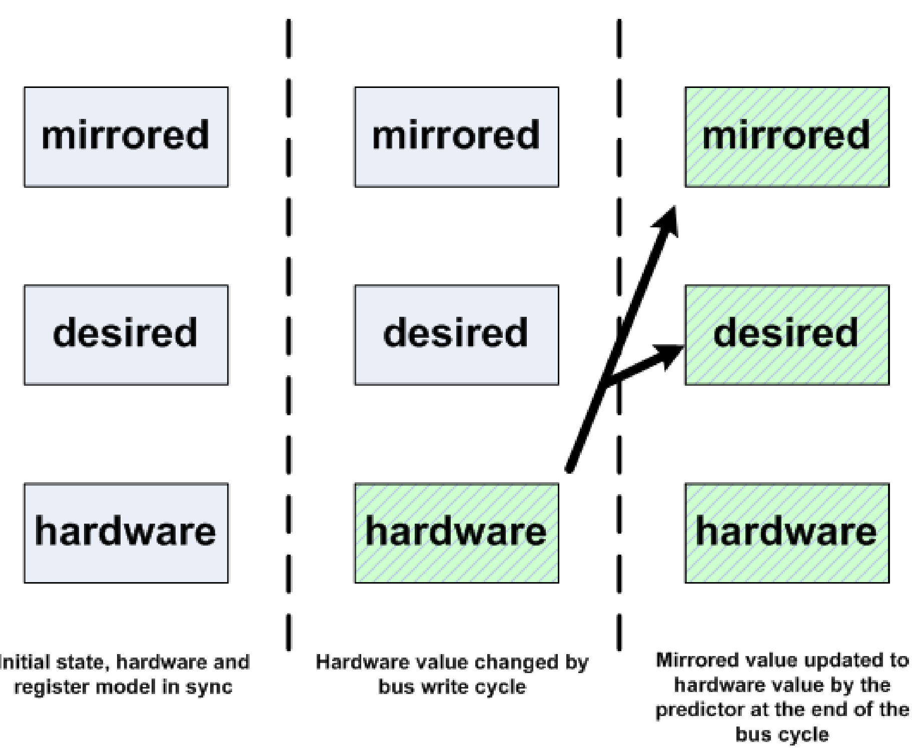
The register model can either use front door or back door access to hardware registers. Front door access involves using the bus interface via an agent and simulates real life bus transfers with their associated timing and event relationships. Back door access uses simulator data base access routines 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.



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_t    value,
           input  uvm_door_e        path = UVM_DEFAULT_DOOR,
           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 AND of the inverse 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	~A & B	If (B != D)	~D
W1S, W1SRC	A	B	A | B	If (B != D)	D
W1T	A	B	A ^ B	If (B != D)	D ^ B
W0C, W0CRS	A	B	A & B	If (B != D)	D

W0S, W0SRC	A	B	~A | B	If (B != D)	~D
W0T	A	B	~A ^ B	If (B != D)	~(D ^ 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 W01 modes, the variable ‘first’ in the Desired Value column has a value of one(1’b1) 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):
```

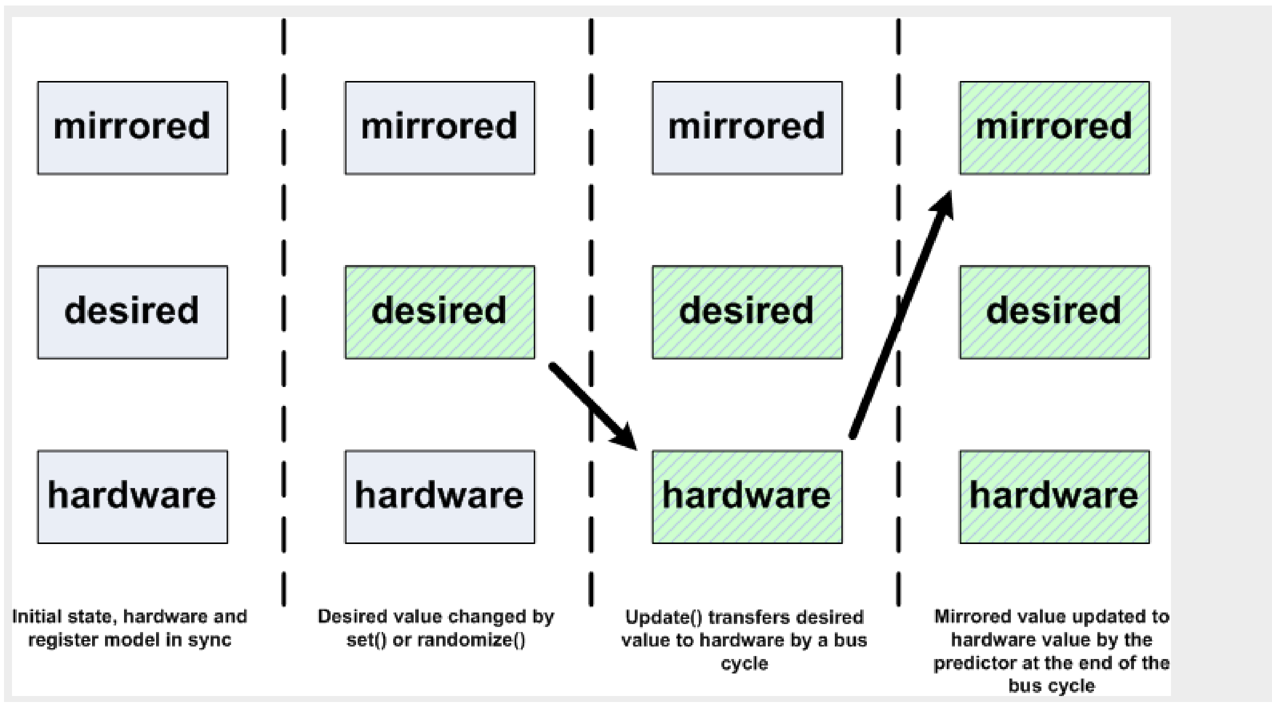
```
spi_rm.ctrl.char_len.set(ctrl_value);
```

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.

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.



```
// 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_door_e         path   = UVM_DEFAULT_DOOR,
            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 summarizes 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_door_e	UVM_DEFAULT_DOOR	To specify whether a front or back door access is to be used - can be UVM_FRONTDOOR, UVM_BACKDOOR, UVM_PREDICT, UVM_DEFAULT_DOOR
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 summaries the various register access methods and the level at which they can be used for back door and front door accesses.

	Front door access			Back door access			
Method	**Block**	**Register**	**Field**	**Block**	**Register**	**Field**	**Comment**
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.

Memory-Level Stimulus

Memory Model Overview

The UVM register model also supports memory access. Memory regions within a DUT are represented by 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_door_e        path = UVM DEFAULT DOOR, //
 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_door_e               path = UVM_DEFAULT_DOOR,
// 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_door_e            path =
UVM_DEFAULT_DOOR, // 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_door_e                  path =
UVM_DEFAULT_DOOR, // 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.

Note: Memory burst accesses are actually broken down into single transfers by the underlying implementation of the register model adaption layer. If you are interested in having to support true burst accesses then you could implement your own extended version of the uvm_mem class, overloading the do_write() and do_read() methods. An alternative approach using the uvm_reg_item is described in the DVCon 2016 paper entitled "A New Class of Registers".

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:
```

```
    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
```

Register Sequence Examples

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(); 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
      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 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: 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.

Built-in Register Sequences

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 summarized 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.

Disclaimer: The built in register sequences do not return a pass or fail status, if you use them, then you need to take responsibility for checking the result. In early bring up testing, this might be done by manually checking the result, in later testing, scoreboarding will be required.

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_reg_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

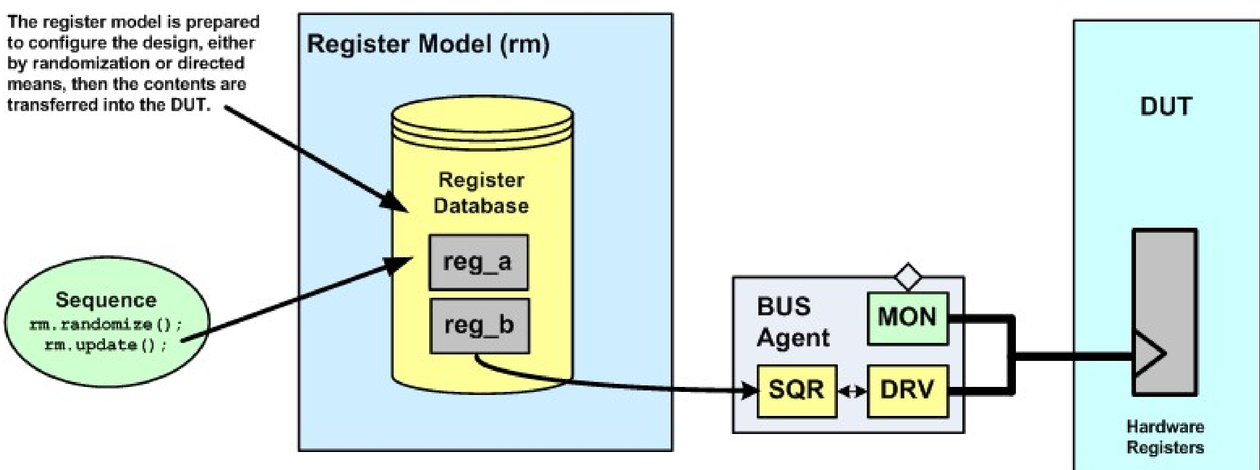
The code for this example can be downloaded via the following link:

Configuring Registers

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 initializes 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 initializing from reset has the shortcoming 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.



Register Configuration Use Model

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 {0, 1,
 2, 4, 8, 16, 32, 64, 128};
                              }) 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.

Register-Level Scoreboards

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 behavior:

- 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 database can be accessed. Field or register values can either be accessed by reference using a path to the mirroed field value (e.g. spi_rm.ctrl.ie.value) or by calling the get_mirrored_value() access method (e.g. spi_rm.ctl.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.

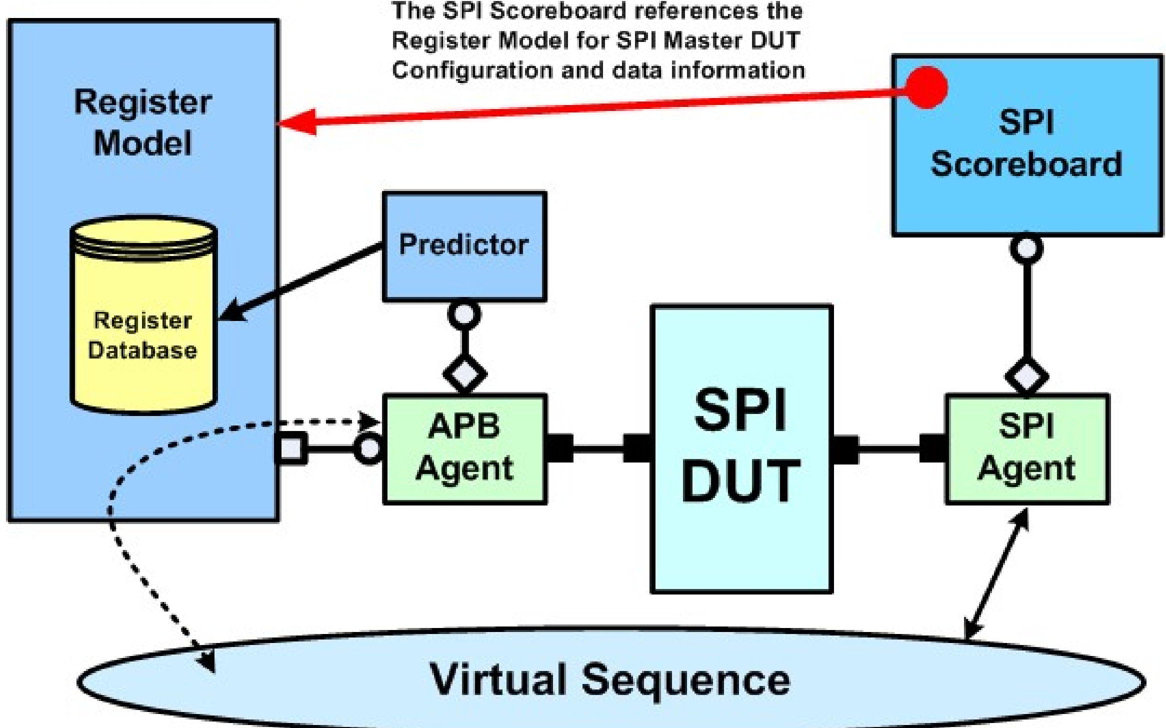
Another way is to use the register model to check the contents of the DUT against their expected contents. For instance, in the case of a serial to parallel converter, the scoreboard could determine the expected contents of the receive buffer, and use a back door peek() access of the hardware receive 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 in the Registers/StimulusAbstraction article.

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 env's 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;
        `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:

Register-Level Functional Coverage

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 specialized 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
     end
endfunction: write
```

Coding Guideline: Wrap covergroups within uvm_objects

A covergroup should be implemented within a wrapper class derived from a 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: sample

endclass: covergroup_wrapper
```

Testbench Acceleration through Co-Emulation

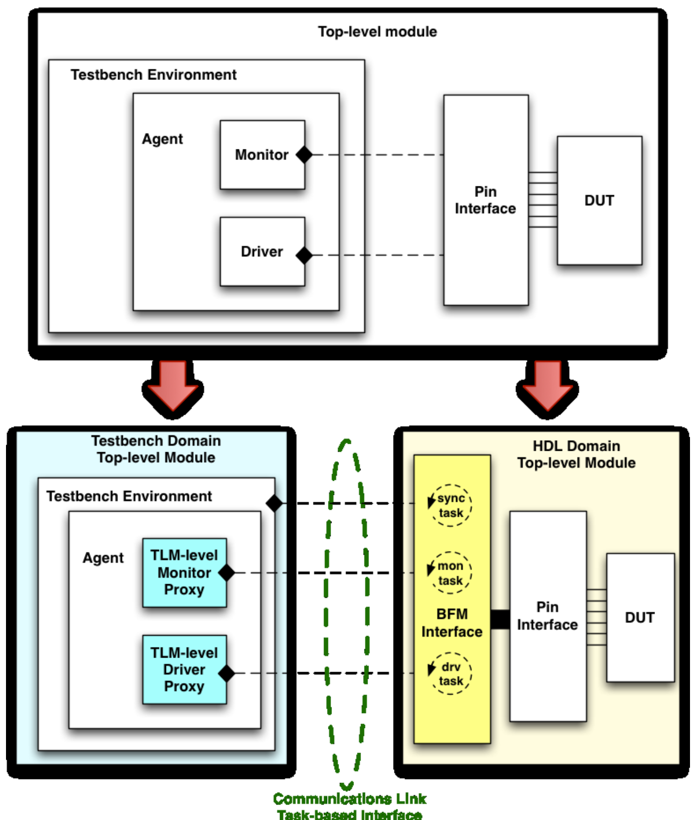
Emulation

Topic Overview

A hardware emulation platform, such as Mentor's 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. Refer to the Mentor 3-series white paper on "Reducing Design Risk with Testbench Acceleration" [1] 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.

Further details can be found in the aforementioned Mentor 3-series white paper [1], providing a more elaborate explanation of UVM testbench acceleration architectural and performance fundamentals and considerations.

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.

To arrive at this goal, different effort levels are required depending on one's starting point. If your testbench was constructed with the testbench architecture advocated and described throughout this UVM cookbook, relatively little effort is needed as the testbench already exhibits the dual TB-HDL domain structure with suitably partitioned transactors. Some additional work remains to ensure that all code in the HDL domain is synthesizable and uses performance optimal constructs such as proxy back-pointers. This entails in particular your driver and monitor transactor state machines, and clock and reset generation logic.

On the other hand, if the starting point is a traditional single top testbench, then significantly more work is required. In that case, the structural partitioning into two separated TB and HDL domains including the pertinent transactor splitting must be done "surgically". In particular, the transactors must be split into an untimed transaction-level proxy component in the TB domain (e.g. a SystemVerilog class like a uvm_component derivative) and a synthesizable BFM in the HDL domain (e.g. a SystemVerilog interface). User-defined tasks and functions - effectively representing transactions - are to be used to implement the transaction-level communication between the proxy and

the BFM [1], across the simulator-emulator boundary. An HDL BFM interface implements synthesizable tasks and functions to be called from the proxy via a virtual interface. Conversely, the BFM interface may initiate function calls back into its proxy via an object handle also referred to as back-pointer.

Lastly, irrespective of starting point, since code in the TB domain must be untimed, any of the aforementioned explicit timing control statements left in the testbench code must be semantically resolved, i.e re-modeled by synchronization with the timed HDL domain through the task-based proxy-BFM communication interface described above, or simply removed where possible.

Worked Example

The methodology pages contain a simple example. Another more comprehensive example starting from a single top level domain further illustrates the methodology and can be found here.

References

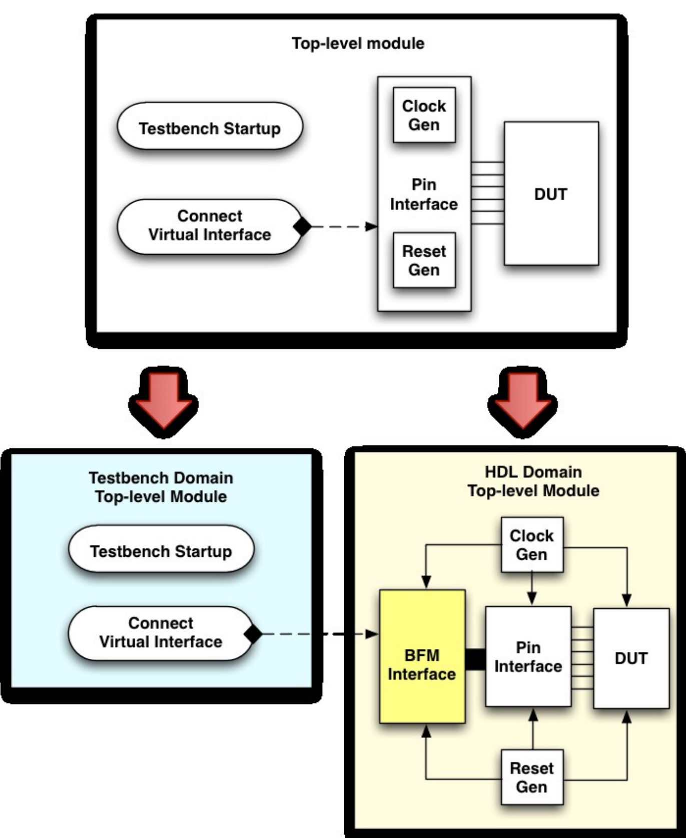
[1] http://go.mentor.com/4xnvq

Separate Top-Level Modules

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 domains, each with its own top-level module in separate files, called the HDL hierarchy and the testbench (TB) hierarchy. This dual-domain methodology generally follows the procedures detailed in the DualTop article and the architecture used throughout this cookbook defined in the testbench architecture article.

The actual use of an emulator imposes two additional related requirements. Firstly, while this cookbook overall does not require that the HDL domain be synthesizable, it becomes a requirement when targeting emulation. Secondly, all logic in the HDL domain must be synchronized to a clock-edge (a.k.a. be clock-synchronous). The remaining verification code residing in the testbench hierarchy executes in the software simulator that must be synchronized with the emulator, and must be untimed (e.g. "@(posedge clk)", "wait(1ns)", and "#10" are not allowed).



This article uses a **single top** bidirectional driver example as a simple testbench to illustrate the required partitioning between synthesizable HDL and untimed testbench domains. If your testbench already follows the architecture espoused throughout this cookbook, then you need to concern yourself only with ensuring that everything under your HDL top level be synthesizable:

```
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

// UVM 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
```

Interface and DUT Instantiation

Clock Gen

Reset Gen

Connect Virtual Interface

Testbench Startup

Testbench Domain HDL Domain

Above is the original top-level code written in typical single-top manner. In addition to the DUT and interface instantiations, it has initial blocks that generate a clock and a reset, and initiate the testbench.

The first thing that needs to be done to support co-emulation is to identify the various code fragments as belonging in either the testbench domain or the HDL domain, and to move the code accordingly into two separate files, one per domain. Each file will contain its own top-level module. The remainder of this article details the code changes in both domains required to support this methodology.

HDL Domain

Here is the complete HDL domain top-level code, with explanation to 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), .gpio(GPIO));

bidirect_bus_driver_bfm DRIVER(.clk        (BUS.clk),       BFM Instantiation
                               .resetn     (BUS.resetn),
                               .addr       (BUS.addr),
                               .write_data (BUS.write_data),
                               .rnw        (BUS.rnw),
                               .valid      (BUS.valid),
                               .ready      (BUS.ready),
                               .read_data  (BUS.read_data),
                               .error      (BUS.error));

// tbx vif_binding_block
initial                                                     Virtual Interface Connection
  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

// Free running clock
// tbx clkgen
initial                                                     Clock Gen
  begin
    clk = 0;
    forever begin
      #10 clk = ~clk;
    end
  end

// Reset
// tbx clkgen
initial                                                     Reset Gen
  begin
    resetn = 0;
    #50 resetn = 1;
  end

endmodule: top_hdl
```

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), .gpio(GPIO));

bidirect_bus_driver_bfm DRIVER(.clk        (BUS.clk),
                               .resetn     (BUS.resetn),
                               .addr       (BUS.addr),
                               .write_data (BUS.write_data),
                               .rnw        (BUS.rnw),
                               .valid      (BUS.valid),
                               .ready      (BUS.ready),
                               .read_data  (BUS.read_data),
                               .error      (BUS.error));
```

Additionally, BFM interfaces should be created to contain the timed portions of the driver and monitor transactors.

```
interface bidirect_bus_driver_bfm (input  bit          clk,
                                   input  bit          resetn,
                                   output logic [31:0] addr,
                                   output logic [31:0] write_data,
                                   output logic        rnw,
                                   output logic        valid,
                                   input  logic        ready,
                                   input  logic [31:0] read data,
                                   input  logic        error);
  // pragma attribute bidirect_bus_driver_bfm partition_interface_xif

  // Synthesizable code implementing the timed transactor portion goes
here ...

endinterface
```

The // pragma is used by the Veloce software to enable extended SystemVerlog code synthesis capabilities (XRTL) and must be applied to BFMs.

Virtual Interface Connection

In the original testbench, the UVM transactors drive DUT signals through a pin-level interface. Veloce does not allow direct access to HDL signals from the testbench domain. 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 either in the testbench domain or the HDL domain. In the testbench domain, a hierarchical cross-reference to the HDL-side interface is used 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, Veloce also conveniently supports virtual interface binding in the synthesizable HDL domain, right where the corresponding concrete BFM interface is instantiated. It requires the use of a virtual interface binding code block along with a simple pragma in adherence with the specific format given below.

```
module top_hdl;

  ...

  bidirect_bus_driver_bfm DRIVER( ... );

  // 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
```

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, the Veloce emulator supports synthesis of a superset of RTL code called XRTL (eXtended RTL). According to 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 applies for the reset block as well. Clock and reset signals must 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 Veloce/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. Veloce 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, with explanation to follow.

```
// Top level test bench module
module top_tb;

import uvm_pkg::*;
import bidirect_bus_pkg::*;

// UVM start up:
initial
  begin
    run_test("bidirect_bus_test");
  end

endmodule: top_tb
```

Testbench Startup

The code that starts the class-based UVM testbench environment remains in the testbench domain.

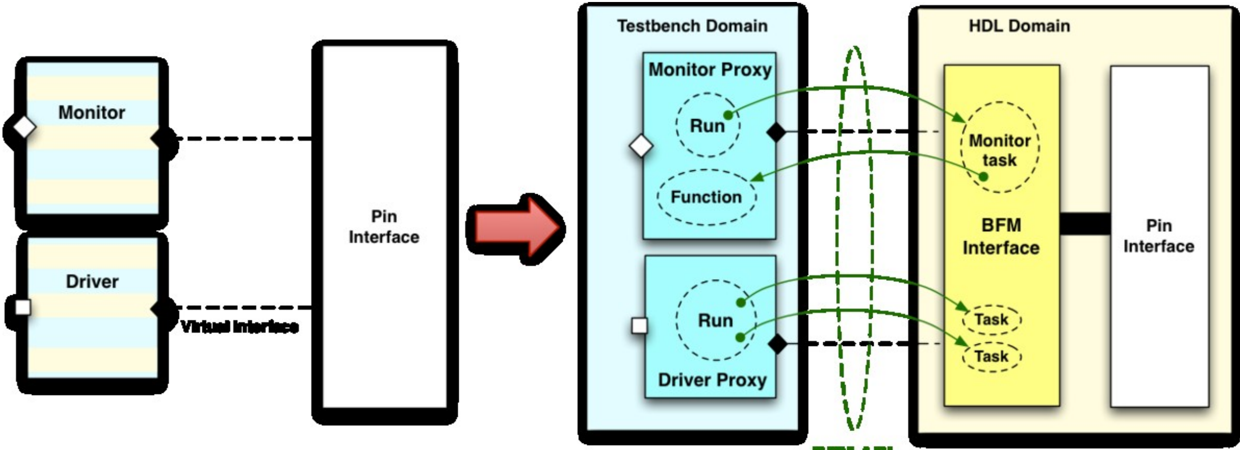
```
run_test("bidirect_bus_test");
```

Shared Parameters

Any parameters that are common 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.

Split Transactors

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 for the top-level module, the portions of transactor code that are timed and access signals must be identied, and separated 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 a sequence item port or an analysis port.

The second point, interacting with DUT signals, needs to be partitioned from the code that sends/receives transactions. This signal-level 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 semantics are modeled 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.

When following the dual domain testbench architecture used throughout this cookbook, the split between proxy and BFM is implied and has been done already. However, there is still work to do because the BFM and all interactions with the BFM must be with synthesizable code. This includes any arguments to functions and tasks that may be called to/from the proxy and BFM. Object handles can no longer be used to pass information. Instead, individual arguments can be used or a packed struct could be used. More information on this topic is available in the article on Defining the BFM API.

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 **single top** 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

endclass: bidirect_bus_driver
```

Testbench Domain HDL Domain

This must thus be split into two partitioned parts inside separate files, with the timed interface protocol ultimately executed on the emulator and the TLM-based portion run 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

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 execute transaction
      // Pass required transaction data to BFM as arguments to the task
      //Get response from BFM through task output 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 article on Defining the BFM API.

HDL Domain Code

The timed, signal-level 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 SystemVerilog XRTL subset. For instance, Veloce XRTL requires task and function arguments to be input or output only - no inout and ref arguments.

```
interface bidirect_bus_driver_bfm (input  bit          clk,
                                   input  bit          resetn,
                                   output logic [31:0] addr,
                                   output logic [31:0] write_data,
                                   output logic        rnw,
                                   output logic        valid,
                                   input  logic        ready,
                                   input  logic [31:0] read_data,
                                   input  logic        error);
// pragma attribute bidirect_bus_driver_bfm partition_interface_xif

import bidirect_bus_shared_pkg::bus_seq_item_s;

initial begin
  // Default conditions:
  valid <= 0;
  rnw <= 1;
end

task wait_for_reset(); // pragma tbx xtf
  @(posedge resetn);
endtask

task do_item(input bus_seq_item_s req, output bus_seq_item_s rsp); // pragma tbx xtf
   @(posedge clk);
   repeat(req.delay-1) begin
     @(posedge clk);
   end
   valid <= 1;
   addr <= req.addr;
   rnw <= req.read_not_write;
   if(req.read_not_write == 0) begin
     write_data <= req.write_data;
   end
   while(ready != 1) begin
     @(posedge 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 = read_data; // If read - copy returned read data
   end
   rsp.error = error; // Copy bus error status
   valid <= 0; // End the pin level bus transaction
endtask

endinterface
```

Pragma Comments

Veloce 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 pragma 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 pragma comment:

```
// pragma tbx xtf
```

Note that Veloce XRTL does not permit these BFM API tasks and functions to be called locally within the HDL domain, so only annotate BFM API routines with this pragma. If you really need to reuse a task or function for such dual purpose, you can often do so by coding it without a pragma for local HDL domain usage, and also putting a call inside a wrapper task/function with a pragma.

Initial Values and Synchronization

The two default value assignments have been moved into an initial block in the BFM interface. Veloce 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 Transactor Code

The original transactor code started with 'repeat (req.delay) begin ...'. Veloce 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 as 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 code section, the attributes of the transaction request (addr, read_not_write, write_data, etc.) are applied to the pin interface. These attributes come from the input argument 'req' of the 'do_item' BFM task, rather than from the initial UVM 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

In the final code section, the response is sent back to the transactor proxy in the testbench domain, via the 'do_item' BFM task output argument 'rsp':

```
// 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
```

Back Pointers

Overview

In the original **single top** bidirectional driver example, 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. Veloce supports "back-pointers" in the HDL domain, which are simply handles (back) 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". The BFM's proxy in the testbench domain merely starts up the BFM thread(s) in its 'run()' task, and the transaction-level communication back from the BFM to the proxy is handled by designated (and appropriately named) functions:

```
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

The back-pointer to the proxy in the testbench domain is declared inside the corresponding BFM interface in the HDL domain:

```
interface bidirect_bus_driver_bfm (...);
// pragma attribute bidirect_bus_driver_bfm partition_interface_xif
...

bidirect_bus_pkg::bidirect_bus_driver proxy;


...
endinterface
```

The BFM's back-pointer is then assigned somewhere after binding the BFM virtual interface [1] but before the start of the UVM run phase. 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 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 (and seemingly wrongful!) a task, so it must be wrapped in a task rather than a function. Since the BFM needs 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

```
interface bidirect_bus_driver_bfm (input  bit          clk,
                                   input  bit          resetn,
                                   output logic [31:0] addr,
                                   output logic [31:0] write_data,
                                   output logic        rnw,
                                   output logic        valid,
                                   input  logic        ready,
                                   input  logic [31:0] read_data,
                                   input  logic        error);
// pragma attribute bidirect_bus_driver_bfm partition_interface_xif

import bidirect_bus_shared_pkg::bus_seq_item_s;
import bidirect_bus_pkg::bidirect_bus_driver;

bidirect_bus_driver proxy;

bus_seq_item_s req;

bit go;

function void run(); // pragma tbx xtf
  // Default conditions:
  valid <= 0;
  rnw <= 1;
  go = 1;
endfunction

initial begin
  // Wait for reset to end and go signal from run()
  wait(resetn & go);
  @(posedge clk);
  forever
    begin
      bit success;
      proxy.try_next_item(req, success); // Start processing req item
      while (!success) begin
        @(posedge clk);
        proxy.try_next_item(req, success); // Start processing req item
      end
      @(posedge clk);
      repeat(req.delay-1) begin
        @(posedge clk);
      end
      valid <= 1;
      addr <= req.addr;
      rnw <= req.read_not_write;
      if(req.read_not_write == 0) begin
        write_data <= req.write_data;
      end
      while(ready != 1) begin
        @(posedge 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 = read_data; // If read - copy returned read data
      end
      req.error = error; // Copy bus error status
      valid <= 0; // End the pin level bus transaction
      proxy.item_done(req); // End of req item
    end
end

endinterface
```

Declaring the Back-Pointer

The declaration of a back-pointer from the BFM interface in the HDL domain to its corresponding proxy in the testbench domain is as shown earlier above [2].

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 [3]. 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
```

References

[1] https://verificationacademy.com/cookbook/Emulation/SeparateTopLevels#Virtual_Interface_Connection
[2] https://verificationacademy.com/cookbook/Emulation/BackPointers#Setting_the_Back-Pointer
[3] https://verificationacademy.com/patterns-library/implementation-patterns/analysis-patterns/bfm-notification-pattern

Defining an API

Overview

As the timed portion of the traditional UVM transactor must be moved over to the HDL domain, the pertinent lines of code must be extracted and rearranged into synthesizable FSMs and associated logic inside a BFM interface, a SystemVerilog interface. This entails in particular the creation of tasks and functions in that BFM interface to effectively define a transaction-based API for the BFM, accessible from the testbench domain. When designing the BFM API, consider that the partitioned testbench and HDL domains are running on separate platforms, workstation and emulator, and therefore communication between them can have a big impact on run-time performance. Consequently, communication should be kept relatively infrequent and data-rich, as opposed to frequency and "data-light" (e.g. imagine a Ferrari driving through a large city's downtown intersections, hitting every red light).

The data involved in the HDL domain to drive and monitor the DUT's pin interfaces is typically associated with stimulus and analysis transactions in the testbench domain. These transactions are generally class-based and model an abstraction of the pin-level DUT data. They are optimized for class-based communication and manipulation. Yet, the HDL domain does not support class objects, and hence transaction data attributes from the testbench domain may need to be converted into synthesizable DUT-compatible data and vice-versa.

BFM Task and Function Arguments

When the cycle-based signal-level 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 **single top** 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 clk);
  repeat(req.delay-1) begin
    @(posedge clk);
  end

  valid <= 1;
  addr <= req.addr;
  rnw <= req.read_not_write;
  if(req.read_not_write == 0) begin
```

```
    write_data <= req.write_data;
  end

  while(ready != 1) begin
    @(posedge 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 = read_data; // If read - copy returned read data
  end
  rsp.error = error; // Copy bus error status

  valid <= 0; // End the pin level bus transaction
endtask
```

This task takes the transaction attributes 'addr', 'read_not_write', 'write_data', and 'delay' from a request transaction, and sets the attributes 'error' and 'read_data' of a response transaction.

If there are just a few data attributes, like address and data, and already in a packed format, these are readily passed as straight individual task arguments. On the other hand, if there are many pertinent transaction attributes, it may be better suited to define a synthesizable packed struct container and pass all attributes together via a single struct task argument. For the current driver example, the BFM requires four request and two response data attributes, so a struct is defined accordingly:

```
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(...)' utilizes this packed struct type to exchange data with its proxy. Note that Veloce supports input and output arguments - inouts are syntactic sugar while refs are a bit like pointers and not generally synthesizable. The 'do_item' task arguments are hence as follows:

```
task do_item(input bus_seq_item_s req, output bus_seq_item_s rsp);
```

Converting Between Testbench Transactions and HDL BFM Data

The 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() thread(s), or indirectly by calling helper functions. For the split proxy-BFM pairs it can still be done this way, i.e. in a BFM's proxy, or alternatively 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 converter class is then used to map the original bus sequence item object into a corresponding struct, and vice versa:

```
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);
```

Example

Overview

This article steps through the process of converting a comprehensive traditional **single top** UVM example testbench to an equivalent one with a dual domain partitioned structure that is ready for co-emulation with Veloce. This emulation-ready testbench adheres to the methodology advocated throughout this cookbook, enabling testbench reuse for both hardware-assisted acceleration and conventional software simulation. For your own testbench, if it exhibits the dual domain architecture from inception, some of the steps described here are not applicable since you have effectively performed them already. Specifically, this includes top level partitioning and transactor splitting. Some work may still be needed to ensure clocks and resets are ready for emulation, and everything in the HDL domain including BFM function and task call arguments are synthesizable.

Split Top Level

The first thing to be done is to partition the single top level of the original code into two separated domain hierarchies [1], the HDL domain for the DUT and other accelerated HDL code, and the testbench domain for, well, the class-based testbench. Click here to see the original single top and corresponding dual top code in full. Most of the original code belongs in the HDL top level module. The exception is the UVM initial block to launch the UVM test, which remains in the testbench domain.

The HDL Domain

The DUT domain top-level module should contain:

- DUT instantiation
- DUT pin interface instantiations
- Clock and reset signal generation
- BFM interface instantiations

Interface Declarations and Instantiations

Transactor BFM interfaces and protocol signal interfaces are instantiated and interconnected in the traditional manner through direct signal port connections.

```
logic PCLK;
logic PRESETn;

apb_if APB (PCLK, PRESETn);
spi_if SPI (PCLK, PRESETn);

apb_driver_bfm  APB_DRIVER(
   .PCLK    (APB.PCLK),
   .PRESETn (APB.PRESETn),
   .PADDR   (APB.PADDR),
   .PRDATA  (APB.PRDATA),
   .PWDATA  (APB.PWDATA),
   .PSEL    (APB.PSEL),
   .PENABLE (APB.PENABLE),
   .PWRITE  (APB.PWRITE),
   .PREADY  (APB.PREADY)
```

```
);
apb_monitor_bfm APB_MONITOR(
   ...
);
spi_driver_bfm  SPI_DRIVER(
   .PCLK    (SPI.PCLK),
   .PRESETn (SPI.PRESETn),
   .clk     (SPI.clk),
   .cs      (SPI.cs),
   .miso    (SPI.miso),
   .mosi    (SPI.mosi)
);
spi_monitor_bfm SPI_MONITOR(
   ...
);
```

For the attentive reader, while one could conceivably do the interconnection more elegantly using interface ports, like so

```
apb_if APB (...);

apb_driver_bfm  APB_DRIVER(APB)
...
```

this would lead to LRM non-compliant usage when virtual BFM interfaces come into play. A pity really, and something to consider for the SystemVerilog standard body. That said, despite the availability of SystemVerilog interfaces, the use of signal ports as DUT module ports is so entrenched in the design community anyway and largely continues to prevail over interface ports.

Clock and Reset Generation

Veloce requires that clock and reset generation be modeled in separate initial blocks, each annotated with the same pragma comment (//tbx clkgen):

```
// 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

With Veloce XRTL support for a so-called "vif binding block" as shown below (note the pragma comment), the HDL domain may effectively contain the (non-synthesizable) UVM-based code to put the various virtual BFM interface handles into the UVM configuration database to set up the virtual interface binding 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. It may also handle the virtual interface binding instead of doing this (more elegantly, but sometimes more restrictively) on the HDL-side as above (don't do it 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

This example boasts two interfaces: an APB interface and a SPI interface. Each interface has a driver and a monitor transactor. These four transactors must each be split into a BFM-proxy pair [1], namely a synthesizable (acceleratable) BFM for the HDL domain and a TLM-level portion (representing the BFM) for the testbench domain. For 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 a coding style using implicit state machines.

Clock Synchronization

Note that all tasks in the APB and SPI BFMs must start by synchronizing to a clock edge, as required by Veloce XRTL.

Communication Format

The separated BFM and proxy of the split transactor can communicate through a BFM task call via a virtual BFM interface handle in the proxy. Veloce requires that the data format of the task arguments is in a packed form. When there is a lot of data to be transferred, a delegate class to perform the conversion between transaction object data and the BFM packed data is convenient:

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:

Data transfer over the SPI interface is rather primitive and hence the use of a conversion class seems overkill. The data is passed directly instead as individual BFM task arguments.

APB Agent

Click here to view the original APB agent code and the resulting split transactors for the driver and monitor.

This is relatively straightforward, putting the timed, signal-level code into tasks in the BFM.

APB Driver

All the signal-level code is moved to a task 'do_item()' of the BFM. The driver proxy calls this task and the transaction data is converted using a delegate class, as discussed above:

```
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 initialization code in the original driver that runs once at the beginning of the run task is placed in an equivalent initial block in the BFM:

```
initial begin
  PSEL <= 0;
  PENABLE <= 0;
  PADDR <= 0;
end
```

APB Monitor

The monitor BFM uses a back-pointer to send the transaction data back to its testbench proxy. The testbench monitor proxy 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 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, a back pointer is declared and named appropriately 'proxy'. When a transaction is observed from pin activity, inside the BFM's run() thread, it is sent back to the testbench by calling a function of the BFM's proxy 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
```

Note that the back-pointer declaration in the monitor BFM is annotated with a Veloce pragma comment:

```
// pragma tbx oneway proxy.write
```

While not required, this pragma comment can be seen as a Veloce compiler directive that is useful under certain circumstances to enable significant so-called one-way caller performance optimization. Effectively, the pragma comment directs the Veloce emulator to keep running, i.e. not halt the design clocks and not yield to testbench threads, at the time of the 'write()' function call, as would otherwise happen for the purpose of communication/synchronization with the simulator. This is feasible in particular because the 'write()' function is a strict one-way function - a void function without output arguments and without "side effects" (like a 'pure' void function in C/C++). The pragma can be used more generally in cases where an 'outbound' function call through the proxy back-pointer to the testbench only results in strict zero time passive (as opposed to reactive) consumption (indeed just like an analysis transaction from a monitor out via an analysis port to a scoreboard or coverage collector). Consult the Veloce user documentation for more information on this and other acceleration performance optimizations available.

SPI Agent

Click here to view the original SPI agent code and the resulting split transactors for the driver and monitor.

The original SPI driver and monitor pose a more significant challenge to split and maintain equivalent behavior. The challenge is not with the methodology per se, however, but specifically with the re-coding of the transactor behavior in an equivalent yet synthesizable manner.

The original SPI components use an interface-specific clock, 'SPI.clk'. The protocol code is sensitive to both edges of this clock. In order to make this Veloce synthesis compliant, all activity must be made syncrhonous to a single edge of a single clock. An auxiliary system clock is introduced for this purpose.

SPI Driver

The original driver starts out with code that does not adhere to the methodology rules (no timing control in the testbench side), **and** is also not synthesizable:

```
while(cs === 8'hxx) begin
  #1;
end
```

The HDL-side BFM interface defines an 'init()' task to implement this in a synthesis-friendly way:

```
task init(); // pragma tbx xtf
  @(negedge PCLK);
  while(cs != 8'hff) @(negedge 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 clk);
end
else begin
  @(negedge clk);
end
```

This is rewritten for synthesis through clock synchronization with the negedge of PCLK, and using a task argument RX_NEG to handle and mimic sensitivity to both edges of the original SPI clk:

```
for(int i = 1; i < num_bits-1; i++) begin
  @(negedge PCLK);                        //--
```

```
  while(clk == RX_NEG) @(negedge PCLK); //   |- mimics if (RX_NEG == 1)
@(posedge clk) else @(negedge clk)
  while(clk != RX_NEG) @(negedge PCLK); //--
  miso <= spi_data[i];
  if(cs == 8'hff) begin
    break;
  end
end
```

SPI Monitor

The SPI monitor is implemented in similar fashion as the APB monitor. A main 'run()' task is implemented in the BFM interface and when a transaction is observed, the BFM calls a 'write()' function in the testbench monitor proxy through a back-pointer. SPI clock synchronization handling is done in the same way as above for the SPI driver, by bringing in 'PCLK' as the system clock.

The main challenge of the monitor is handling the fork/join_any in the original code:

```
fork
  begin
    while(cs != 8'hff) begin
      @(clk);
      if(clk == 1) begin
        item.nedge_mosi[p] = mosi;
        item.nedge_miso[p] = miso;
        p++;
      end
      else begin
        item.pedge_mosi[n] = mosi;
        item.pedge_miso[n] = miso;
        n++;
      end
    end
  end
  begin
    @(clk);
    @(cs);
  end
join_any
disable fork;
```

There are two processes forked, one that collects bits from the serial stream and the other 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 = clk;                             // --
@(negedge PCLK);                           //    |- mimics @(clk);
while(clk == clk_val) @(negedge PCLK); // --
```

```
while(cs == local_cs) begin
  if(clk == 1) begin
    nedge_mosi[p] <= mosi;
    nedge_miso[p] <= miso;
    p++;
  end
  else begin
    pedge_mosi[n] <= mosi;
    pedge_miso[n] <= miso;
    n++;
  end
  clk_val = clk;              // ---
  @(negedge PCLK);            //     |
  while(clk == clk_val) begin //     |- mimics @(clk) with premature
break on cs change
    @(negedge PCLK);          //     |
    if (cs != local_cs) break;      // ---
  end
end
```

Remove Timing from Testbench Domain

The run_phase() task of the original SPI test class is as follows:

```
task spi_test::run_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: run_phase
```

This code has a pound-delay timing control statement that violates the methodology rules. Such delays in the test or in sequences can be remodeled using 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
```

In order to call this task from any point in the testbench, one 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
    `uvm_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
    `uvm_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::run_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: run_phase
```

Example Driver

Original 'Single Top' 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
      `uvm_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

```
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
      `uvm_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 (input  bit          PCLK,
                          input  bit          PRESETn,
                          output logic [31:0] PADDR,
                          input  logic [31:0] PRDATA,
                          output logic [31:0] PWDATA,
                          output logic [15:0] PSEL, // Only connect the
 ones that are needed
                          output logic        PENABLE,
                          output logic        PWRITE,
                          input  logic        PREADY);
// pragma attribute apb_driver_bfm partition_interface_xif

import apb_shared_pkg::apb_seq_item_s;

initial begin
  PSEL <= 0;
  PENABLE <= 0;
  PADDR <= 0;
end

task do_item(apb_seq_item_s req, int psel_index, output apb_seq_item_s
rsp); // pragma tbx xtf
  @(posedge PCLK);
  repeat(req.delay-1) // ok since delay is constrained to be between 1
and 20
    @(posedge PCLK);

  rsp = req;
  rsp.error = (psel_index < 0);
```

```
  if(rsp.error) return;

  PSEL[psel_index] <= 1;
  PADDR <= req.addr;
  PWDATA <= req.data;
  PWRITE <= req.we;
  @(posedge PCLK);
  PENABLE <= 1;
  while (!PREADY)
    @(posedge PCLK);
  if(PWRITE == 0)
  begin
    rsp.data = PRDATA;
  end
  PSEL <= 0;
  PENABLE <= 0;
  PADDR <= 0;
endtask: do_item

endinterface: apb_driver_bfm
```

Original 'Single Top' 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());
      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(input bit           PCLK,
                          input bit           PRESETn,
                          input logic [31:0] PADDR,
                          input logic [31:0] PRDATA,
                          input logic [31:0] PWDATA,
                          input logic [15:0] PSEL,
                          input logic         PENABLE,
                          input logic         PWRITE,
                          input logic         PREADY);
// 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 PCLK);

  forever begin
    // Detect the protocol event on the TBAI virtual interface
    @(posedge PCLK);
    if(PREADY && PSEL[index]) // index identifies PSEL line this
monitor is connected to

    // Assign the relevant values to the analysis item fields
      begin
```

```
        item.addr = PADDR;
        item.we = PWRITE;
        if(PWRITE)
          begin
            item.data = PWDATA;
          end
        else
          begin
            item.data = PRDATA;
          end
        // Publish the item to the subscribers
        proxy.write(item);
      end
  end
endtask: run

endinterface: apb_monitor_bfm
```

Example Agent

Original 'Single Top' 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 (input  bit         PCLK,
                          input  bit         PRESETn,
                          input  logic       clk,
                          input  logic [7:0] cs,
                          output logic       miso,
                          input  logic       mosi);
// pragma attribute spi_driver_bfm partition_interface_xif

initial begin
  miso = 1;
end

task init(); // pragma tbx xtf
  @(negedge PCLK);
  while(cs != 8'hff) @(negedge 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 PCLK);

  while(cs == 8'hff) @(negedge PCLK);

  num_bits = no_bits;
  if(num_bits == 0) begin
    num_bits = 128;
  end
  miso <= spi_data[0];
  for(int i = 1; i < num_bits-1; i++) begin
   @(negedge PCLK);                          \
    while(clk == RX_NEG) @(negedge PCLK);   > // mimics if (RX_NEG == 1)
@(posedge clk) else @(negedge clk)
    while(clk != RX_NEG) @(negedge PCLK); /
    miso <= spi_data[i];
    if(cs == 8'hff) begin
      break;
    end
  end
endtask: do_item

endinterface: spi_driver_bfm
```

Original'Single Top' 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
      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(input bit         PCLK,
                          input bit         PRESETn,
                          input logic       clk,
                          input logic [7:0] cs,
                          input logic       miso,
                          input logic       mosi);
// 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] local_cs;
  int n;
  int p;
  bit clk_val;

  @(negedge PCLK);

  while(cs != 8'hff) @(negedge PCLK);

  forever begin
    while(cs == 8'hff) @(negedge PCLK);

    n = 0;
    p = 0;
    nedge_mosi <= 0;
    pedge_mosi <= 0;
```

```
    nedge_miso <= 0;
    pedge_miso <= 0;
    local_cs <= cs;

    clk_val = clk;                          \
    @(negedge PCLK);                         > // mimics @(clk);
    while(clk == clk_val) @(negedge PCLK); /

    while(cs == local_cs) begin
      if(clk == 1) begin
        nedge_mosi[p] <= mosi;
        nedge_miso[p] <= miso;
        p++;
      end
      else begin
        pedge_mosi[n] <= mosi;
        pedge_miso[n] <= miso;
        n++;
      end
      clk_val = clk;              \
      @(negedge PCLK);             \
      while(clk == clk_val) begin    > // mimics @(clk) with premature
break on cs change
        @(negedge PCLK);           /
        if (cs != local_cs) break; /
      end
    end

    // Publish to the subscribers
    proxy.write(nedge_mosi, pedge_mosi, nedge_miso, pedge_miso,
local_cs);
  end
endtask: run

endinterface: spi_monitor_bfm
```

Example Top-Level Model

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
  PRESETn = 1;
  forever begin
    #10ns PCLK = ~PCLK;
  end
end

endmodule: top_tb
```

HDL Domain Top-Level

```
module top_hdl;

`include "timescale.v"

// 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

// 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);
end

apb_driver_bfm  APB_DRIVER(.PCLK(APB.PCLK),
                           .PRESETn(APB.PRESETn),
                           .PADDR(APB.PADDR),
                           .PRDATA(APB.PRDATA),
                           .PWDATA(APB.PWDATA),
                           .PSEL(APB.PSEL),
                           .PENABLE(APB.PENABLE),
                           .PWRITE(APB.PWRITE),
                           .PREADY(APB.PREADY));
apb_monitor_bfm APB_MONITOR(.PCLK(APB.PCLK),
                            .PRESETn(APB.PRESETn),
                            .PADDR(APB.PADDR),
                            .PRDATA(APB.PRDATA),
```

```
                                        .PWDATA(APB.PWDATA),
                                         .PSEL(APB.PSEL),
                                        .PENABLE(APB.PENABLE),
                                         .PWRITE(APB.PWRITE),
                                        .PREADY(APB.PREADY));

// 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);
                 uvm_config_db #(virtual apb_monitor_bfm)::set(null, "uvm_test_top",
$psprintf("%m.APB_MONITOR") , APB_MONITOR);
end

spi_driver_bfm        SPI_DRIVER(.PCLK(SPI.PCLK),
                                        .PRESETn(SPI.PRESETn),
                                         .clk(SPI.clk),
                                         .cs(SPI.cs),
                                         .miso(SPI.miso),
                                       .mosi(SPI.mosi));
spi_monitor_bfm SPI_MONITOR(.PCLK(SPI.PCLK),
                                         .PRESETn(SPI.PRESETn),
                                          .clk(SPI.clk),
                                          .cs(SPI.cs),
                                          .miso(SPI.miso),
                                          .mosi(SPI.mosi));

// tbx vif_binding_block
initial begin
   import uvm_pkg::uvm_config_db;
                 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(
      // 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)
);


//
// 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
```

Testbench Domain Top-level

```
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

UVM Debugging

Built-In Debug

While simulators should provide (and do provide in the case of QuestaSim [1]) 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.

Note that the majority of the features described in this article are not documented in the the IEEE 1800.2 LRM, however they are supported as debug features of the Accellera implementation of the UVM 1800.2 class library. The features described are available in earlier versions of the UVM (1.1a through to 1.2).

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-1.2/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-1.2/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 *recurse* is set, then configuration information for all children and below are printed as well. if *audit* 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 *recurse* 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 be 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 similiar 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.

nction	Prototype	Description
	`<font size="7.62">function void print ( int all_types = 1 )</font>`	Prints the state of the uvm_factory, includi registered types, inst overrides, and type When *all_types* is 0, and instance override displayed. When *all_* (default), all registere user-defined types ar as well, provided the names associated wit When *all_types* is 2, types (prefixed with included in the list of registered types.

eate_by_type	**function void** debug_create_by_type (uvm_object_wrapper requested_type, **string** parent_inst_path = "", **string** name = "")	These methods perfo same search algorith create_* methods, bu not create new object Instead, they provide information about wh of object it would ret listing each override applied to arrive at th When requesting by *requested_type* is a h the type's proxy obje Preregistration is not When requesting by *request_type_name* representing the requ type, which must hav registered with the fa with that name prior request. If the factory recognize the requested_type_nam is produced and a nul returned. If the optional *parent_inst_path* is then the concatenatio {parent_inst_path, ".",~name~}, forms a instance path (contex used to search for an override. The *parent_inst_path* is obtained by calling th uvm_component::get on the parent.
ate_by_name	**function void** debug_create_by_name (**string** requested_type_name, **string** parent_inst_path = "", **string** name = "")	

The most commonly used of these function is print. That is used in the form factory.print() which returns information similar 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:

```
uvm printfactory
```

NOTE: In UVM1.2, the factory cannot be referenced directly in the uvm_pkg. Instead, it must be accessed via

```
uvm_factory::get();
```

.

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-1.2/src/base/uvm_phase.svh(1114) @ 0 ns: reporter
[PH/TRC/STRT] Phase 'common.run' (id=93) Starting phase
# UVM_INFO ../uvm-1.2/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-1.2/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'uvm' (id=142) Starting phase
# UVM_INFO ../uvm-1.2/src/base/uvm_phase.svh(1342) @ 0 ns: reporter [PH/TRC/DONE] Phase
'uvm' (id=142) Completed phase
# UVM_INFO ../uvm-1.2/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-1.2/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched' (id=154) Starting phase
# UVM_INFO ../uvm-1.2/src/base/uvm_phase.svh(1342) @ 0 ns: reporter [PH/TRC/DONE] Phase
'uvm.uvm_sched' (id=154) Completed phase
# UVM_INFO ../uvm-1.2/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-1.2/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-1.2/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-1.2/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-1.2/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-1.2/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-1.2/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched.reset' (id=184) Starting phase
# UVM_INFO ../uvm-1.2/src/base/uvm_phase.svh(1342) @ 80 ns: reporter [PH/TRC/DONE] Phase
'uvm.uvm_sched.reset' (id=184) Completed phase
# UVM_INFO ../uvm-1.2/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-1.2/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-1.2/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-1.2/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 similar 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 get_connected_to() (usually used with ports) and get_provided_to() (usually used with imps or exports).

Note: The get_connected_to() and get_provided_to() functions were renamed in the IEEE 1800.2. In previous versions of the UVM, the two calls were named debug_connected_to() and debug_provided_to(), their functionality is as described here.

Function	Prototype	Description
get_connected_to	`function void get_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.
get_provided_to	`function void get_provided_to ( int level = 0, int max_level = -1 )`	The get_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 get_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}].get_connected_to
```

When get_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
[get_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 get_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}].get_provided_to
```

When get_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 [get_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.

Funtion	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 similiar 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.

(Note: print_topology() is documented in the IEEE 1800.2 LRM)

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 functon 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 UVM_HIGH 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 thier verbosity settings altered by using the +uvm_set_verbosity plusarg.

PlusArg	Description
+UVM_VERBOSITY=<uvm_verbosity>	This will set the default verbosity of all the components created in the UVM testbench to the specified verbosity.
+uvm_set_verbosity=<component>,<id>,<verbosity>,<phase>	This will set the verbosity of a specific component for a given UVM phase
+uvm_set_verbosity=<component>,<id>,<verbosity>,time,<time>	This will set the verbosity of a specific component from a specific time in the simulation.

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.

References

[1] http://www.mentor.com/products/fv/questa/
[2] http://events.dvcon.org/2012/proceedings/papers/12_3.pdf

Using Verbosity for Debug

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_NONE	Displayed	Displayed	Displayed	Displayed	Displayed
UVM_LOW	Filtered	Displayed	Displayed	Displayed	Displayed
UVM_MEDIUM	Filtered	Filtered	Displayed	Displayed	Displayed
UVM_HIGH	Filtered	Filtered	Filtered	Displayed	Displayed
UVM_FULL	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 root 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_hier(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 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 noticable when called repeatedly.  With a
verbosity setting of UVM_HIGH, this $sformat() 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
$sformat() 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", $sformatf("%s, data[%0d] = 0x%08x",
name, ii, data[ii]), UVM_HIGH);
```

If uvm_report_info() was used directly instead

```
uvm_report_info("Message_ID", $sformatf("%s, data[%0d] = 0x%08x", name,
 ii, data[ii]), UVM_HIGH);
```

the $sformat() string would have be processed before entering the uvm_report_info() function even through ultimately that string will not be printed due to the UVM_HIGH verbosity setting.

Command-Line Processor

Introduction

The UVM command line processor is used to 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.

The IEEE 1800.2 LRM defines a sub-set of the built-in UVM plusargs, the Accellera UVM 1800.2 implementation supports all the plusargs, most importantly the ones that enable debug features in the library.

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

Plusargs documented in the IEEE 1800.2 LRM:

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 interger 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

Plusargs not documented in the IEEE 1800.2 LRM, but available in the Accellera UVM implementations

Plusarg Name	Description	Example Usage
+UVM_TIMEOUT	+UVM_TIMEOUT=<timeout>,<overridable> allows users to change the global timeout of the UVM framework. The timeout value is specified as an interger 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_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

All of these plusargs are documented in the IEEE 1800.2 LRM

e	Description	Example Usage
	+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,**tim**
	+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 | separated list of the other UVM message actions (UVM_DISPLAY, UVM_LOG, UVM_COUNT, UVM_EXIT, UVM_CALL_HOOK, UVM_STOP). **Note**: As of UVM 1.1, this plusarg can only take a single UVM message action. You can not have a | seperated 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_ACTIO
	+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_WARN
ERRIDE VERRIDE	+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
_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 employees to denote user defined plusargs created by Mentor, A Siemens Business.

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 woudl 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:

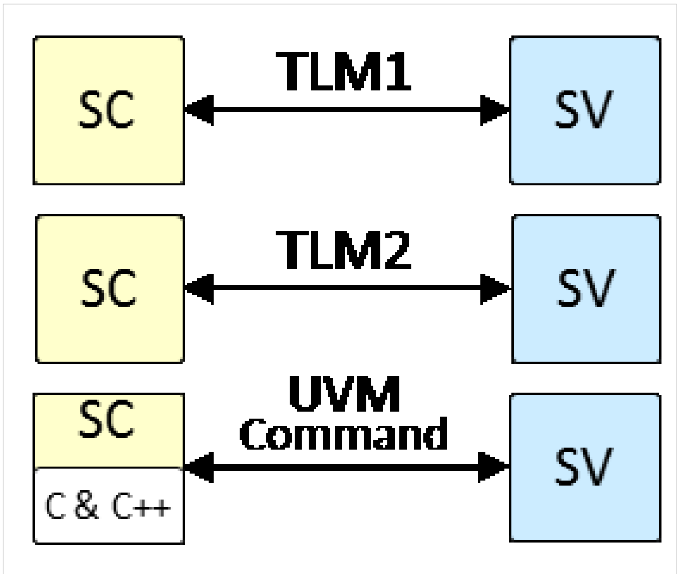
```
string split_values[$]
foreach (my_values[ii]) begin
  uvm_split_string(my_values[ii], ",", split_values);

  if (split_values[1] == "YES") begin
    ...
  end
end
```

UVM Connect - SV-SystemC interoperability

UVM Connect(UVMC)

Topic Overview

The UVM Connect library 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

UVM Connect (a.k.a. UVMC) enables the following use models, all designed to maximize IP reuse:

- Abstraction Refinement - Reuse your SystemC architectural models as reference models in SystemVerilog-UVM verification. Reuse your stimulus generation agents in SystemVerilog (a.k.a. SV) to verify models in SystemC (a.k.a. SC).
- 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. And 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, raise and drop objections to phase transitions, set and get configuration, issue UVM-style formated reports, set report filters, print UVM topology, set factory type and instance overrides, and more.

The UVM Connect library makes connecting TLM models in SystemC and UVM in SystemVerilog a relatively straightforward process. However, because UVM Connect is effectively integrating several technologies, you'll need to have basic knowledge of SystemC, SystemVerilog, and the UVM, and TLM standards. Refer to the *References* section in the UVMC-Primer [1] for a partial list of relevant documentation. You may also wish to read the brief *TLM Review* section included in the same document.

Key Features

This section enumerates some important characteristics 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 the free, open-source Accellera UVM 1.1d and later.
- *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-enabling independently designed models to 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.

Further Information

For quick overview about UVMC's *Connections*, *Conversion*, and *Command API* features, please see those chapters of this cookbook following the links at the top of this page.

For a comprehensive user guide and documentation of examples about UVM-Connect see the **UVM-Connect Primer** [1] for the *.pdf* form or here [2] for the *HTML* form.

And for a download of all source code for the UVMC library and examples documented in the primer, follow this link,

Finally, the following two links have yet more resources about UVM-Connect including training presentations, videos, links to both .pdf and HTML forms of the UVM-Primer,

- UVM Connect Overview [3]
- UVM Connect Course [4]

References

[1] https://verificationacademy.com/resource/64187
[2] https://verificationacademy.com/verification-methodology-reference/uvmc-2.3.1/docs/html/index.html
[3] https://verificationacademy.com/topics/verification-methodology/uvm-connect
[4] https://verificationacademy.com/courses/uvm-connect

UVMC 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:

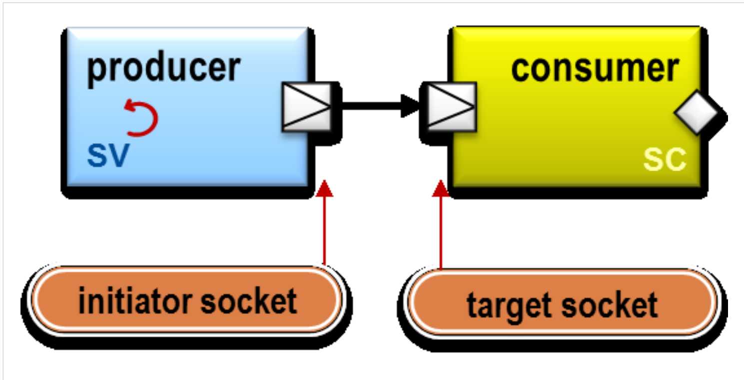
```
uvmc_tlm #(trans)::connect (port_handle, "lookup");
uvmc_tlm #(trans)::connect_hier (port_handle, "lookup");
```

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.)

Further Information

For more comprehensive information specifically about UVMC's *Connections* feature, please see that chapter of the **UVM-Connect Primer** [1] for the *.pdf* form or here [2] for the *HTML* form.

And for a download of all source code for the UVMC library and examples documented here and in the primer, follow this link

UVMC Conversion Layer

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.

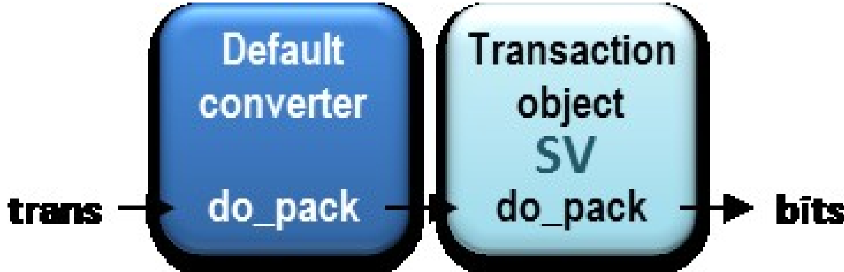
If, however, you are not using the TLM GP, the task of creating a converter in SC and SV is fairly simple. Let's say we have a transaction with a command, address, and variable-length data members. The basic definitions in SV and SC might look like this:

```
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.



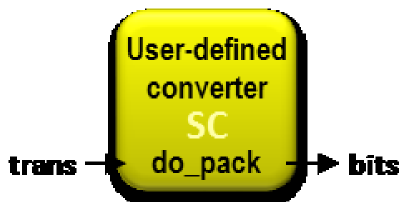
UVM transaction conversion is usually implemented in its *do_pack()* and *do_unpack()* methods, or via the `*uvm_field* macros. The *do_pack()/unpack()* method implementations have better performance and provide greater flexibility, debug, and type-support than the `*uvm_field* macros. The extra time 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* and `*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.

Further Information

For more comprehensive information specifically about UVMC's *Conversion* feature, please see the *Converters* chapter of the **UVM-Connect Primer** [1] for the *.pdf* form or here [2] for the *HTML* form.

And for a download of all source code for the UVMC library and examples documented here and in the primer, follow this link

UVMC Command API

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();
```

Further Information

For more comprehensive information specifically about UVMC's *Command API* feature, please see the *UVM Commands* chapter of the **UVM-Connect Primer** [1] for the *.pdf* form or here [2] for the *HTML* form.

And for a download of all source code for the UVMC library and examples documented here and in the primer, follow this link

UVM Versions and Compatibility

UVM1.2/Summary

Accellera UVM 1.2 release information

Cookbook topics which link to this page are affected by backwards compatibility issues or migration issues, when the Accellera UVM1.2 release is used. Check the categories below for sources of information and advice, and a summary of the changes involved in migrating from UVM1.1x to UVM1.2.

Configuration

The set_config_[int, string, object]() and get_config_[int, string, object]() methods in UVM1.1 are deprecated in UVM1.2. As recommended in the UVM Coding Guidelines, you should always use the uvm_config_db::[set,get]() API instead. UVM1.2 includes some convenience typedefs for the uvm_config_db as follows:

```
typedef uvm_config_db#(uvm_bitstream_t) uvm_config_int;
typedef uvm_config_db#(string) uvm_config_string;
typedef uvm_config_db#(uvm_object) uvm_config_object;
typedef uvm_config_db#(uvm_object_wrapper) uvm_config_wrapper;
```

We do not recommend using uvm_config_int, since the uvm_bitstream_t type is 4096 bits long. It is better to use the actual type of the value being configured.

Care must be taken when switching from set_config_object() and get_config_object() to uvm_config_db#(uvm_object)::[set,get](), as the clone functionality of the old API no longer exists. This is **not** handled by the uvm11-to-uvm12.pl conversion script.

Core Services

UVM contains a number of features, such as the Factory and the Report Server, that UVM1.2 has grouped under the name "Core Services." Rather than refer to these individual elements directly, as was done in UVM1.1, the uvm_coreservices_t singleton provides an API to refer to these elements. First, you must get a handle to the uvm_coreservices_t singleton:

```
uvm_coreservice_t  cs = uvm_coreservice_t::get();
```

Now you can use the cs handle to get access to the factory or report_server:

```
uvm_factory f = cs.get_factory();
```

Since we recommend that factory calls be done through the uvm_component API, gaining access to the uvm_factory in this way is hardly ever necessary. Similarly, there is almost never a need to access the uvm_report_server either. However, the uvm_report_server may be extended to customize message output. In UVM1.2, the base rerport server functionality is actually in a class called uvm_default_report_server, so if you wish to extend the server, you'll need to extend from the default implementation:

```
class my_server extends uvm_default_report_server;
...
```

There really isn't any need to extend the factory either, but if you needed to, you would similarly extend from uvm_default_factory.

Extending uvm_factory and uvm_report_server

In UVM1.1, the uvm_factory and uvm_report_server base classes could be extended directly for the user to customize. In UVM1.2, these base classes are abstract classes, and the "base class" functionality is built into a "default" class extended from the abstract base class:

```
class uvm_default_factory extends uvm_factory;
  ...
endclass

class uvm_default_report_server extends uvm_report_server;
  ...
endclass
```

Therefore, any UVM1.1 user code that extends either uvm_factory or uvm_report_server must be updated to extend the default implementation of each in UVM1.2.

Factory Overrides

Normally, factory overrides are used intentionally to change the type of a component returned from the factory. The typical flow is to create a test extended from a base test and specify the override in the extended test. As such, the verification engineer should always know what specific type is to be returned when objects or components are created. The syntax for a Factory override by type is:

```
// <original_type>::type_id::set_type_override(<substitute_type>::get_type(), replace);
```

In UVM1.1x, if <substitute_type> and <original_type> were the same, nothing would happen and any previous override would continue to be in force. In UVM1.2, specifying the same type for both <original_type> and <substitute_type> would remove any pending override for <original_type>. There should never be a time when a well-written testbench would have to use this feature.

Factory References

In UVM1.1, the factory is a singleton that can be referenced directly via

```
uvm_pkg::factory
```

. In UVM1.2, the factory no longer exists directly in the package. Rather, it must be accessed via

```
fact=uvm_factory::get();
```

However, the factory is not usually referenced directly, but instead is accessed through the uvm_component's API. As shown in BuiltInDebug, the factory debug methods can be accessed via the Questa command line. The factory should rarely, if ever, be accessed from within user code.

Messaging

In UVM1.2, the messaging macros have been expanded to treat messages as objects. The use of the standard messaging macros hasn't changed:

```
`uvm_info(ID, MSG, VERBOSITY)
`uvm_warning(ID, MSG)
`uvm_error(ID, MSG)
`uvm_fatal(ID, MSG)
`uvm_info_context(ID, MSG, VERBOSITY, RO)  //RO is a uvm_report_object
`uvm_warning_context(ID, MSG, RO)
`uvm_error_context(ID, MSG, RO)
`uvm_fatal_context(ID, MSG,RO)
```

We recommend that you continue to use these macros for messaging in UVM, regardless of the version you are using. However, should you wish to take advantage of the new ability, UVM1.2 provides a series of begin-end macros to create message objects of the same severities:

```
`uvm_info_begin(ID, MSG, VERBOSITY, RM =__uvm_msg)
`uvm_info_end

`uvm_warning_begin(ID, MSG, RM =__uvm_msg)
`uvm_warning_end

`uvm_error_begin(ID, MSG, RM =__uvm_msg)
`uvm_error_end

`uvm_fatal_begin(ID, MSG, RM =__uvm_msg)
`uvm_fatal_end

`uvm_info_context_begin(ID, MSG, VERBOSITY, RO, RM =__uvm_msg)
`uvm_info_context_end

`uvm_warning_context_begin(ID, MSG, RO, RM =__uvm_msg)
`uvm_warning_context_end

`uvm_error_context_begin(ID, MSG, RO, RM =__uvm_msg)
`uvm_error_context_end

`uvm_fatal_context_begin(ID, MSG, RO, RM =__uvm_msg)
`uvm_fatal_context_end
```

Since these new message objects have an ID (in Verbosity, in the case of uvm_info) associated with them, the set_report_action and other messaging functions will continue to work on them. However, it is now possible to add fields to the message object, through the use of additional macros:

```
`uvm_message_add_tag(NAME, VALUE, ACTION=(UVM_LOG|UVM_RM_RECORD))
`uvm_message_add_int(VAR, RADIX, LABEL = "",
ACTION=(UVM_LOG|UVM_RM_RECORD))
`uvm_message_add_string(VAR, LABEL = "",
ACTION=(UVM_LOG|UVM_RM_RECORD))
```

```
`uvm_message_add_object(VAR, LABEL = "",
ACTION=(UVM_LOG|UVM_RM_RECORD))
```

Notice that each addition to the message object allows for its own action to be specified. It is unlikely that a testbench would need to specify different actions on individual parts of a message, but the possibility now exists.

Objections

UVM1.2 adds a set_propagate_mode() method to the uvm_objection object to allow objection raising and lowering to bypass all intermediate steps in the hierarchy and propagate immediately up to the uvm_top. As recommended in the UVM Coding Guidelines, objections should only be raised and dropped from the test, so there are no intermediate layers between the test and uvm_top. There should never be a time when a well-written testbench would have to use this feature.

In addition, UVM1.2 adds a set_automatic_phase_objection() method to the uvm_sequence object, enabling the sequence to raise an objection when it starts and drop the objection when it completes, in violation of the UVM Coding Guidelines. This method should not be used.

Sequence Arbitration

In a backward-incompatible change, UVM1.2 changes the element names for the uvm_sequencer_arb_mode enum from

```
SEQ_ARB_FIFO (Default)
SEQ_ARB_WEIGHTED
SEQ_ARB_RANDOM
SEQ_ARB_STRICT_FIFO
SEQ_ARB_STRICT_RANDOM
SEQ_ARB_USER
```

to

```
UVM_SEQ_ARB_FIFO (Default)
UVM_SEQ_ARB_WEIGHTED
UVM_SEQ_ARB_RANDOM
UVM_SEQ_ARB_STRICT_FIFO
UVM_SEQ_ARB_STRICT_RANDOM
UVM_SEQ_ARB_USER
```

Transaction Recording

Transaction recording is performed in UVM through the use of macros, to allow tool/vendor-specific implementations to write information into a database. In UVM1.1, the macros are:

```
`uvm_record_attribute // never call directly
`uvm_record_field
```

Similar to messaging, the "field" variant allows you to specify a name/value pair to record. UVM1.2 adds several more macros for recording specific types:

```
`uvm_record_int
`uvm_record_string
`uvm_record_time
`uvm_record_real
```

At the macro level, the UVM1.2 versions of uvm_record_[attribute, field] are backward compatible with UVM1.1d. If you wish to record an element of a specific type, use the new type-specific macros.

UVM1800.2/Summary

The UVM became the IEEE Standard for Universal Verification Methodology Language Reference Manual - 1800.2 in 2017. The IEEE 1800.2 LRM was based on the content of UVM 1.2, but there are differences. This article summarizes those differences, but the definitive guide is the IEEE 1800.2 LRM.

The IEEE publishes standards and therefore does not develop base class libraries. The Accellera standards body who developed the UVM as it evolved has developed a UVM 1800.2 implementation that is available as an open source library.

The IEEE standardization process brings stability to the UVM. IEEE standards tend to change very slowly with updates emerging every 5 years or so. However, the standardization process did present an opportunity to rationalize the UVM API and the Accellera code base. As a user, this means that when you write any new code for UVM 1800.2 you should only use API calls or variables that are documented in the LRM. If you use any other methods or variables then you run the risk that these will either change or be removed as the library implementation is refactored.

The Accellera UVM 1800.2 implementation

The Accellera standards organization publishes an open source base class library for IEEE 1800.2 UVM. This contains the 1800.2 API and a compile switch called UVM_ENABLE_DEPRECATED_API to allow you to compile and simulate with the UVM 1.2 API to help with migration from UVM 1.2 to UVM 1800.2.

The Accellera implementation also contains some debug methods that come from UVM 1.2 but are not in the IEEE LRM. These methods are documented by Accellera in their release.

The Accellera code base also contains some code that is neither documented in the 1800.2 LRM nor is it deprecated. This type of code may consist of internal code required to implement the published API, or it may be some legacy UVM code that may or may not become standard in the future. The overriding guideline is only to use what is documented in the 1800.2 LRM.

The Accellera UVM 1800.2 implementation is bundled with Questa and a built-in pre-compiled library is available as part of the Questa installation.

Although the Accellera UVM 1800.2 implementation is open source, you should resist the temptation to look at the source code and use methods or class members not documented in the LRM.

Migrating to UVM 1800.2

If you have existing UVM code that you need to port to UVM 1800.2, then you should first migrate your code to work with UVM 1.2. If you are migrating from a UVM 1.1x version, then please refer to the UVM1.2 summary article for details of the main changes.

The changes between UVM 1.2 and UVM 1800.2 are mainly about rationalization with the introduction of a few new features. The following are the important differences:

Deprecations

In the IEEE 1800.2 the following classes have been deprecated:

- uvm_utils
- uvm_port_component_base
- uvm_port_component
- uvm_sequencer_param_base
- uvm_random_stimulus
- uvm_in_order_comparator
- uvm_in_order_built_in_comparator
- uvm_in_order_class_comparator

These classes are rarely used in practice.

Many methods were deprecated or renamed in an attempt to streamline the code and provide a logical API. There are too many deprecations to list here, but where they are likely to be relevant to user code we mention them in later sections.

However, there is one major deprecation that may impact UVM legacy concerning the use of OVM phase names. In previous versions of the UVM, the OVM phase methods were honored and their use was aliased with the UVM phases. In 1800.2, the OVM phase names are deprecated. This means that if you have OVM phases in your code, then you should migrate them to the UVM phases:

```
// OVM phases                          Equivalent UVM phases:
function void build();                 function void build phase(uvm phase
 phase);
function void connect();               function void
connect phase(uvm phase phase);
function void end_of_elaboration(); function void
end_of_elaboration_phase(uvm_phase phase);
task run();                            task run_phase(uvm_phase phase);
function void report();                function void
report phase(uvm phase phase);
```

uvm_object

All of the uvm_object field methods (copy(), compare(), print(), pack(), unpack()) now take a policy object as an argument. The change here is that the copy() method takes a uvm_copier policy whereas earlier it did not.

The uvm_object has an additional do_execute_op() method that is a user definable hook method called by the policy classes. This method requires the uvm_field_op class which can be used to either replace or augment the various do_copy, do_compare() etc methods via the policy. The do_execute_op() method is called before the do_copy(), do_compare() etc methods.

Messaging

The following classes were introduced in UVM 1.2 and are in the Accellera code, but are not in the 1800.2 LRM:

- uvm_report_message_element_base
- uvm_report_message_int_element
- uvm_report_message_string_element
- uvm_report_message_object_element
- uvm_report_message_element_container

These were concerned with operating on parts of a message string after a call from within a UVM component or object, which seems redundant. We recommend that these classes are not used. There are also some methods associated with message_elements in the uvm_report_object and the uvm_report_catcher, which again are not documented and therefore should not be used:

```
// uvm_report_object and uvm_report_catcher methods not in 1800.2
get_element_container()
add_int()
add_string()
add_object()
// Methods deprecated in uvm_report_catcher:
get_report_catcher()
print_catcher()
```

The Factory

In 1800.2 the factory has been extended to support abstract classes. What this means is that virtual classes can be registered with the factory. Since virtual classes cannot be constructed, this type of factory entry can only be used with an override of a class that provides a non-abstract extension of the abstract class. If you call a type_id::create() method on an abstract class it will result in an error.

This capability seems somewhat redundant, and may even be detrimental to the performance of UVM testbenches. We recommend that this feature not be used.

These are macros associated with the registration of abstract classes with the factory:

```
`uvm_object_abstract_utils()
`uvm_object_abstract_param_utils()
`uvm_component_abstract_utils()
`uvm_component_abstract_param_utils()
```

Event and Barrier Pools

There are two new pool classes that can be used to store collections of events and barriers. Both are specializations of the uvm_pool class.

```
uvm_event_pool
uvm_barrier_pool
```

These pools can either be used as global pools or as more localized pools.

Sequences

The uvm_sequence_base class has been made a virtual class. This may affect sequence code that relied on creating a uvm_sequence_base as a polymorphic handle to call methods in a derived sequence.

The uvm_sequencer_param_base class has been deleted, and its functionality has been merged with the uvm_sequencer_base class. This change should not affect most users unless they have extended the uvm_sequencer_param_base class.

The `uvm_do() macros have been rationalised, so that there are fewer macros but with more arguments. We do not recommend the use of these macros in any case.

Policies

All UVM policy classes now extend from a common uvm_policy class that contains a base set of methods for handling stack objects. The uvm_policy class also allows an extension object to be added to a uvm_policy class, this can be used to customize the policy.

1800.2 introduces a uvm_copier policy object, for use with the copy() method.

All of the policy classes have additional accessor methods for setting and getting policy members. The policy classes now have additional support for traversing object hierarchies.

In practice, these changes do not affect default policy behavior. However, they have the potential to make the policy classes more flexible.

Registers

The uvm_path_e enumerated type has been renamed to uvm_door_e, and the value that is used as the default path variable in the UVM register access methods has changed from UVM_DEFAULT_PATH to UVM_DEFAULT_DOOR. This will impact user code that extends any of these methods. For instance, any register uvm_reg class extensions may be impacted if they extend the read() or write() methods.

In order to facilitate the support of dynamic address maps, the uvm_reg_block and the uvm_reg_map can now be unlocked, and reg_maps and registers can be unregistered and re-added at different addresses before re-locking the uvm_reg_block again. For more details see the article on complex address mapping in the register chapter.

The uvm_reg_item now has accessor (set()/get()) methods for its various variables.

Miscellaneous

There are many APIs where integers were used as argument types, these have now been changed to int since that is a two state variable. For example:

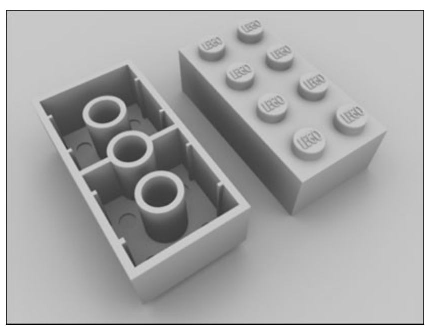
```
// UVM 1.2 integer type used
function integer begin_child_tr(time begin_time = 0, integer
parent_handle = 0);

// UVM 1800.2 int type used
function int begin_child_tr(time begin_time = 0, int parent_handle =
0);
```

Appendix - Deployment

UVC/UVMVerificationComponent

UVM Verification Component

A UVC (**U**VM **V**erification **C**omponent) 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.

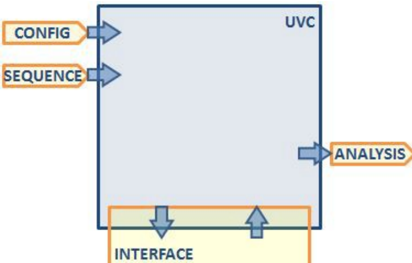
Relevant to native verification of the protocol

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.



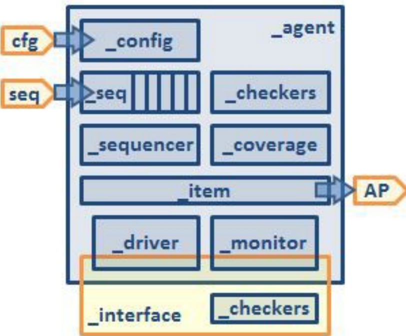
Some protocols will demand architectural departures from the familiar norm; some underlying technology may deviate from it by necessity of the business model relying on protected code, or special non-SV implementations, but familiarity from the user's point of view should be preserved if possible, often 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



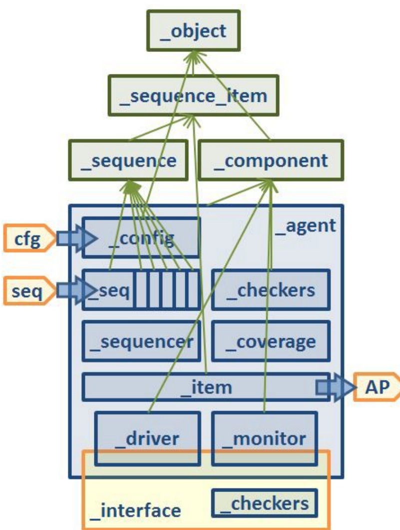
The architecture of a Protocol UVC is intended to include the following elements to match the requirements of behavior:

- 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 it's 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.

```
."
ms (transactions)
ig object
  //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_phase(uvm_phase phase);
    ...
    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_phase(uvm_phase phase);
    ...
    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
```

```
// 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
```

Data & Sequences

Driver

```
// my_driver.svh
```

```
// my_monitor.svh

class my_monitor extends uvm_monitor;
  my_config cfg;
  uvm_analysis_port #(my_item) o_my;

  function void build_phase(uvm_phase phase);
    o_my = new("o_my",this);
  endfunction

  task run_phase(uvm_phase phase);
    forever begin
      my_item it;
      ... monitor it
      o_my.write(it);
    end
  endtask
endclass
```

Monitor

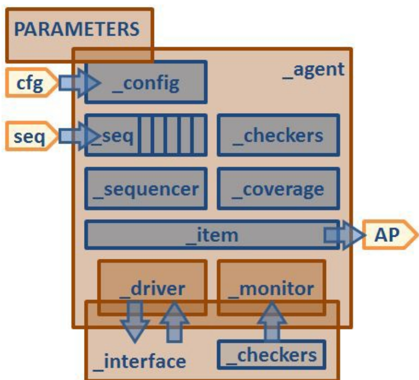
Coverage

```
// 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.



Agent:

```
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:

```
interface my_interface #(type P=int) (clock, reset);
  ...
endinterface
```

Driver:

```
class my_driver #(type params=int) extends uvm_driver #(my_item);
  ...
endclass
```

Monitor:

```
class my_monitor #(type params=int) extends uvm_monitor;
  ...
endclass
```

Coverage Object:

```
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

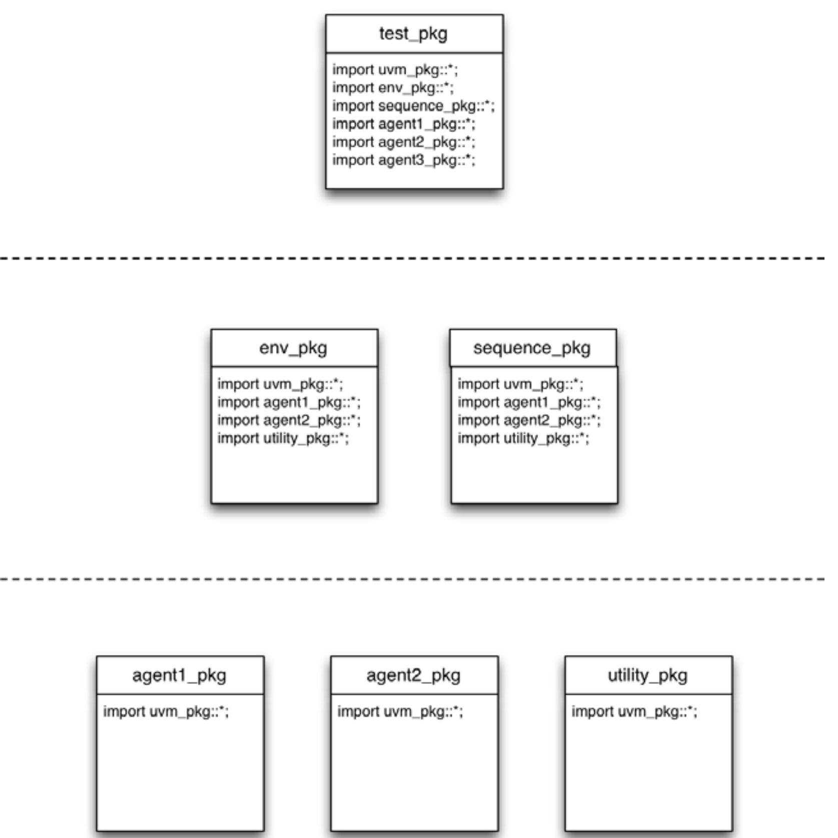
Attributes of a UVC

- relevant to native verification of the protocol or function in question
- familiar artifact for teams who know UVM to integrate, modify, replicate, reuse.
- conforms 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 addtion 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 laid 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 laid 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::agen1_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 [1] which provides additional information and examples.

"Questa can't find the type of a class I'm using."

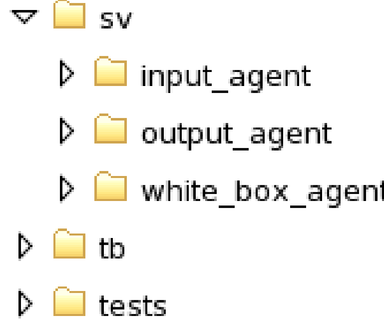
More than likely the package that the class was `included into wasn't imported into the package or module where the class type was trying to be used. Simply add an import statement to the top of the package or module to expose the class definition to the scope of the package or module.

"run_test() can't find my test. I know it's in my test_pkg."

Even if the 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.



References

[1] http://blogs.mentor.com/verificationhorizons/blog/2010/07/13/package-import-versus-include/

Questa/CompilingUVM

Introduction

The UVM class library is an open source SystemVerilog package that relies on DPI c code in order to implement some of the library features such as regular expression matching and register backdoor accesses.

For UVM, we recommend using the built-in, pre-compiled UVM and DPI compiled libraries that ship with Questa. This will remove the need to install any compilers or create a "build" environment. Depending on the Questa version there should be at least 2-3 versions of the UVM library available (1.1d, 1.2, 1800.2)

Simulating the UVM Out-Of-The-Box with Questa

UVM can be used out of the box with Questa very easily. A precompiled, auto-loading version of UVM is available with every release after Questa 10.0a including 10.0a. This includes the SystemVerilog package and the DPI shared object. If any of your code imports the uvm_pkg (import uvm_pkg::*;), then these will be loaded automatically for you.

If you have a file called hello.sv which imports the uvm_pkg, then your flow would look like this:

```
vlib work
vlog hello.sv
vsim hello ...
```

Notice that we don't have to specify +incdir+$UVM_HOME/src or add a -sv_lib command to the vsim command to load the uvm_dpi shared object.

Controlling UVM Versions

Each release of Questa comes with multiple versions of the UVM pre-compiled and ready to load. By default, a fresh install of Questa will load the latest version of UVM that is available in the release. If an older version of UVM is needed, this version can be selected in one of two ways.

Modify the modelsim.ini File

Inside the modelsim.ini file, it contains a line which defines a library mapping for Questa. That line is the mtiUvm line. It looks something like this:

```
mtiUvm = $MODEL_TECH/../uvm-1.1d
```

This example is pointing to the UVM 1.1d release included inside the Questa release. If we wanted to upgrade to UVM 1.2, then we would simply modify the line to look like this:

```
mtiUvm = $MODEL_TECH/../uvm-1.2
```

Command Line Switch

The Questa commands can also accept a switch on the command line to tell it which libraries to look for. This switch overrides what is specified in the modelsim.ini file if there is a conflict. The switch is '-L'. If this switch is used, then all Questa commands with the exception of vlib will need to use the switch.

```
vlib work
vlog hello.sv -L $QUESTA_HOME/uvm-1.2
vsim hello -L $QUESTA_HOME/uvm-1.2 ...
```

Compiling UVM Automatically with 'vlog'

To compile the UVM library automatically with 'vlog' requires a few extra steps. The uvm_pkg must be compiled along with the uvm_dpi shared object. Additionally, +incdir+$UVM_HOME/src must be added to any vlog command which compiles files which `include any uvm files such as uvm_macros.svh. You would need to compile the UVM library with 'vlog' if you want to download the UVM library manually, if a different set of defines is being applied, if there are user defined patches or possibly for political reasons.

To compile the same hello.sv file while also compiling the uvm_pkg manually, then the flow would look like this:

```
vlib work
# Compile your own UVM
vlog +incdir+$UVM_HOME/src $UVM_HOME/src/uvm_pkg.sv
$UVM_HOME/src/dpi/uvm_dpi.cc
# Compile your testbench code
vlog +incdir+$UVM_HOME/src hello.sv
vsim hello ...
```

Notice that we are specifying the dpi c code on a command line directly to vlog. Vlog handles compiling and automatically adding a -sv_lib switch to the vsim command line for us.

Questa Debug Package

Questa includes a SystemVerilog package which enables UVM specific debug features. This package is called questa_uvm_pkg. If the UVM library is compiled with vlog, then the questa_uvm_pkg must also be compiled with vlog. This can be done by adding another vlog command after the uvm_pkg compile.

```
vlib work
# Compile your own UVM
vlog +incdir+$UVM_HOME/src $UVM_HOME/src/uvm_pkg.sv
$UVM_HOME/src/dpi/uvm_dpi.cc
# Compile the questa_uvm_pkg
vlog +incdir+$UVM_HOME/src
+incdir+$QUESTA_HOME/verilog_src/questa_uvm_pkg-1.2/src \
  $QUESTA_HOME/verilog_src/questa_uvm_pkg-1.2/src/questa_uvm_pkg.sv
# Compile your testbench code
vlog +incdir+$UVM_HOME/src hello.sv
vsim hello ...
```

The questa_uvm_pkg must be compiled because it references the uvm_pkg which was compiled with vlog.

Appendix - Coding Guidelines

SV/Guidelines

Mentor, A Siemens Business SystemVerilog Guidelines

SystemVerilog Do's
• Use a consistent coding style - see guidelines • Use a descriptive typedef for variables • Use an end label for methods, classes and packages • Use `includes to compile classes into packages • Define classes within packages • Define one class per file • Only `include a file in one package • Import packages to reference their contents • Check that $cast() calls complete successfully • Check that randomize() calls complete succesfully • Use if rather than assert to check the status of method calls • Wrap covergroups in class objects • Only sample covergroups using the sample() method • Label covergroup coverpoints and crosses
SystemVerilog Don'ts
• Avoid `including the same class in multiple locations • Avoid placing code in $unit • Avoid using associative arrays with a wildcard index • Avoid using #0 delays • Don't rely on static initialization order

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 [1]

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.

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.

Only have one declaration or statement per line. This makes the code clearer and easier to understand and debug.

Not Recommended

ze="7.28">// Variable definition:
able;
mpleted;
_progress;
 	
ments:
next Guideline for the use of begin-end pairs
conditional statements)
 	
== 0)
gress = 1;
 	
gress = 0;
	// Variable definition:
	logic enable, completed, in_progress;
	
	// Statements:
	if(enable == 0) in_progress = 1; else in_progress =

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.

	Not Recommended

e="6.91">// Both statements executed conditionally:
begin
current_count;
= current_target;
>
	if(i > 0) count =
	current_count;
	target = current_target; // This statement is
	unconditionally

This makes the code easier to read and avoids mistakes due to operator precedence issues.

Recommended	Not Recommended

// Boolean or conditional expression
if((A==B) && (B > (C*2)) || (B > ((D**2)+1))) begin
...
end
	// Boolean or conditional expression
	if(A==B && B > C*2 || B > D**2+1) begin
	...
	end

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.

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.

Not Recommended

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  mbus seq item rhs ;

  if(!$cast(rhs_, rhs)) begin
      `uvm_error("do_compare", "cast failed, check type compatability")
      return 0;
  end
  do_compare = super.do_compare(rhs, comparer) && (MADDR == rhs_.MADDR)
 && (MWDATA == rhs_.MWDATA) && (MREAD == rhs_.MREAD) && (MOPCODE ==
rhs_.MOPCODE) && (MPHASE == rhs_.MPHASE) && (MRESP == rhs_.MRESP) &&
(MRDATA == rhs_.MRDATA);
endfunction: do compare
```

Recommended	Code
Recommended	`function bit do_compare(uvm_object rhs, uvm_comparer comparer);` `  mbus seq item rhs ;` `  if(!$cast(rhs_, rhs)) begin` ` ``uvm_error("do_compare", "cast failed, check type compatibility")` ` return 0;` ` end` ` do_compare = super.do_compare(rhs, comparer) &&` ` (MADDR == rhs_.MADDR) &&` ` (MWDATA == rhs_.MWDATA) &&` ` (MREAD == rhs_.MREAD) &&` ` (MOPCODE == rhs_.MOPCODE) &&` ` (MPHASE == rhs_.MPHASE) &&` ` (MRESP == rhs_.MRESP) &&` ` (MRDATA == rhs_.MRDATA);` `endfunction: do compare`

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

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 a 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

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;
```

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
```

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

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".

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.

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

Lower case names are easier to type.

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:

This makes it easier to maintain the code, since it is obvious where the code is for each class.

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

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 shoudl be organized in a hierarchy.

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

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"
```

To compile the package itself, you use a +incdir to reference the source directory. Make sure that there are no hardcoded paths in the path string for the `included file.

```
vlog +incdir+$ABC_PKG/src abc_pkg.sv
```

To compile code that uses the package, you also use a +incdir 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

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
```

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 {[o:32'hF000_FC00]};})
begin
  `uvm_error("seq_name", "randomization failure, please check
constraints")
end
```

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.

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.

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]);
```

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.

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 know to cause problems

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_h;
  endfunction
endclass : A
class B;
  A a_h;
```

```
  protected function new;
    a_h = get_a();
  endfunction
  static function B get_b();
    if (b_top == null) b_top =new();
    return b_top;
  endfunction
endclass : B
```

Covergroups

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.

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;
  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

Sample a covergroup by calling the sample routine, this allows precise control on when the sampling takes place.

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;
```

Name your bins, do not rely on auto-naming.

```
bin minimum_val = {min};
```

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 [2]
- Chris Spear's SystemVerilog for Verification [3]
- Doulos' SystemVerilog Golden Reference Guide [4]
- Adam Erickson's Are Macros Evil? DVCon 2011 Best Paper [1]

References

[1] http://verificationacademy.com/resource/6646

[2] http://www.amazon.com/SystemVerilog-Design-Second-Hardware-Modeling/dp/1441941258

[3] https://www.amazon.com/SystemVerilog-Verification-Learning-Testbench-Language/dp/144194561X

[4] http://www.doulos.com/content/products/golden_reference_guides.php#Anchor-SystemVerilo-43475

SV/PerformanceGuidelines

These guidelines are aimed at enabling you to identify coding idioms that are likely to affect testbench performance. Please note that a number of these guidelines run counter to other recommended coding practices and a balanced view of the trade off between performance and methodology needs to be made.

Whilst some of the code structures highlighted might be recognized and optimized out by a compiler, this may not always be the case due to the side effects of supporting debug, interactions with PLI code and so on. Therefore, there is almost always a benefit associated with re-factoring code along the lines suggested.

SystemVerilog shares many common characteristics with mainstream software languages such as C, C++ and Java, and some of the guidelines presented here would be relevant to those languages as well. However, SystemVerilog has some unique capabilities and short-comings which might cause the unwary user to create low performance and memory hungry code without realizing it.

Tuning the performance of a testbench is made much easier the use of code profiling tools. A code profile can identify 'hot-spots' in the code, and if these places can be refactored the testbench is almost invariably improved. In the absence of a profiling tool, visual code inspection is required but this takes time and concentration. These guidelines are intended to be used before coding starts, and for reviewing code in the light of code profiling or by manual inspection.

Code Profiling

Code profiling is an automatic technique that can be used during a simulation run to give you an idea of where the 'hot-spots' are in the testbench code. Running a code profile is a run time option, which if available, will be documented in the simulator user guide. See the "Profiling Performance and Memory Use" chapter in the Questa User Guide for more information.

When your testbench code has reached a reasonable state of maturity and you are able to reliably run testcases, then it is always worth running the profiling tool. Most code profilers are based on sampling; they periodically record which lines of code are active and which procedural calls are in progress at a given point in time. In order to get a statistically meaningful result, they need to be run for a long enough time to collect a representative sample of the code activity.

In a well written testbench with no performance problems, the outcome of the sampling will be a flat distribution across the testbench code. However, if the analysis shows that a particular area of the testbench is showing up in a disproportionate number of samples then it generally points to a potential problem with that code.

Profiling is an analysis technique and the results will be affected by:

- The characteristics of the testcase(s) that are being analyzed
- Random seeding - causing different levels of activity in the testbench
- Dominant behavior in your testbench - some areas of the testbench may simply be doing more work
- DUT coding style
- The sample interval
- The length of the simulation time that the profile is run for
- What is going on in the simulation whilst the profile is being run

With constrained random testbenches it is always worth running through alternative testcases with different seeds whilst analyzing the profiling report since these may throw light on different coding issues.

Loop Guidelines

Loop performance is determined by:

- The work that goes on within the loop
- The checks that are made in the loop to determine whether it should be active or not

The work that goes on within the loop should be kept to a minimum, and the checks made on the loop bounds should have a minimum overhead. Here are some examples of good and bad loop practices:

Lower Performance Version

```
// dynamic array, unknown size
int array[];
int total = 0;

for(int i = 0; i < array.size(); i++) begin
  total += array[i];
end
```

Higher Performance Version

```
// dynamic array, unknown size
int array[];

int array_size;
int total = 0;

array_size = array.size();

for(int i = 0; i < array_size; i++) begin
  total += array[i];
end
```

Setting a variable to the size of the array before the loop starts saves the overhead of calculating the array.size() on every iteration.

Lower Performance Version

```
int decision_weights[string]; // Assoc
int case_exponents[string];
int total_weights;

foreach(decision_weights[i]) begin
  total_weights += decision_weights[i] *
    case_exponents["high"];
end
```

Higher Performance Version

```
int decision_weights[string]; // Assoc
int case_exponents[string];
int total_weights;
int case_exponent;

case_exp = case_exponents["high"]

foreach(decision_weights[i]) begin
  total_weights += decision_weights[i] *
    case_exp;
end
```

The foreach() loop construct is typically higher performance than for(int i = 0; i < <val>; i++) for smaller arrays.

The lookup of the exponent value in the associative array on every loop iteration is unnecessary, since it can be looked up at the beginning of the loop.

Lower Performance Version

```
int an_array[50];
int indirection_index;
int to_find = 42;

indirection_index = -1;

// Look up an index via the array:
foreach(an_array[i]) begin
  if(an_array[i] == to_find) begin
    indirection_index = i;
  end
end
```

Higher Performance Version

```
int an_array[50];
int indirection_index;
int to_find = 42;

indirection_index = -1;

// Look up an index via the array:
foreach(an_array[i]) begin
  if(an_array[i] == to_find) begin
    indirection_index = i;
    break;
  end
end
```

In this example, an array with unique entries is being searched within a loop for a given value. Using break in the second example terminates the evaluation of the loop as soon as a match is found.

Decision Guidelines

When making a decision on a logical or arithmetic basis there are a number of optimizations that can help improve performance:

Short-circuit logic expressions

The evaluation of a short circuit logic expression is abandoned as soon as one of its elements is found to be false. Using a short-circuit logic express has the potential to speed up a decision. Ordering the terms in a short-circuit expression can also avoid an expensive call if it is not neccessary. Some examples:

With an AND evaluation, if the the first term of the expression is untrue, the rest of the evaluation is skipped:

```
if(A && B && C) begin
  // do something
end
```

With an OR evaluation, if the first term of the expression is true, then the rest of the evaluation is skipped:

```
if(A || B || C) begin
  // do something
end
```

If the terms in the expression have a different level of "expense", then the terms should be ordered to compute the least expensive first:

Lower Performance Version

```
if(B.size() > 0) begin
  if(B[$] == 42) begin
    if(A) begin
        // do something
    end
  end
end
```

Higher Performance Version

```
if(A && (B.size() > 0) && B[$] == 42) begin
  // do something
end
```

If the inexpensive expression A evaluates untrue, then the other expensive conditional tests do not need to be made.

Lower Performance Version

```
if((A||B) && C) begin
  // do something
end
```

Higher Performance Version

```
if(C && (A||B)) begin
  // do something
end
```

A slightly less obvious variant, which saves the computation required to arrive at a decision if C is not true.

Refactoring logical decision logic

Sometimes a little bit of boolean algebra combined with some short-circuiting can reduce the computation involved.

lower performance code

```
if((A && C) || (A && D)) begin
 // do something
end
```

higher performance code

```
if(A && (C || D)) begin
 // do something
end
```

In the above example, refactoring the boolean condition removes one logical operation, using A as a short-circuit potentially reduces the active decision logic

Refactoring arithmetic decision logic

Remembering to refactor arithmetic terms can also lead to optimizations. This does not just apply to decision logic, but also to computing variables.

Lower Performance Version

```
if(((A*B) - (A*C)) > E) begin
  // do something
end
```

Higher Performance Version

```
if((A*(B - C)) > E) begin
  // do something
end
```

In the above example, refactoring avoids a multiplication operation.

Priority encoding

If you know the relative frequency of conditions in a decision tree, move the most frequently occurring conditions to the top of the tree. This most frequently applies to case statements and nested ifs.

Lower Performance Version

```
// Case options follow the natural order:
case(char_state)
  START_BIT: // do_something to start tracking the char (once per word)
  TRANS_BIT: // do something to follow the char bit value (many times
per word)
  PARITY_BIT: // Check parity (once per word, optional)
  STOP_BIT: // Check stop bit (once per word)
endcase
```

Higher Performance Version

```
// case options follow order of likely occurrence:
case(char_state)
  TRANS_BIT: // do something to follow the char bit value (many times
per word)
  START_BIT: // do_something to start tracking the char (once per word)
  STOP_BIT: // Check stop bit (once per word)
  PARITY_BIT: // Check parity (once per word, optional)
endcase
```

Most of the time, the case statement exits after one check saving further comparisons.

Lower Performance Version

```
// ready is not valid most of the time
// read cycles predominate
//
if(write_cycle) begin
  if(addr inside {[2000:10000]}) begin
    if(ready) begin
    // do something
    end
  end
end
else if(read_cycle) begin
  if(ready) begin
    // do something
  end
end
```

Higher Performance Version

```
// ready is not valid most of the time
// read cycles predominate
//
if(ready) begin
  if(read_cycle) begin
    // do something
  end
  else begin
    if(addr inside {[2000:10000]}) begin
      // do something
    end
  end
end
```

In the higher performance version of this example, if ready is not valid, the rest of the code does not get evaluated. Then the read_cycle check is made, which removes the need for the write_cycle check.

Task and Function Call Guidelines

In-Lining Code

In some situations it may be better to re-factor code that is calling sub-routine methods so that the contents of the method are unrolled and put in-line rather than using the sub-routine. This will be particularly true if the sub-routine is relatively short and has multiple arguments.

Task And Functional Call Argument Passing

In SystemVerilog, passing arguments to/from task and function calls is done by making a copy of the variable at the start of the task or function call and then copying back the result of any changes made during the execution of the method. This can become quite an overhead if the arguments are complex variable types such as strings or arrays, and the alternative is to use a reference. Using a reference saves the overhead of the copies but it does mean that since the variable is not copied into the function if it is updated in the task or function, then it is also updated in the calling method. One way to avoid this issue is to make the variable a const ref, this effectively makes it a read only reference from the point of view of the function.

Lower Performance Version

```
function void do_it(input int q[$], input string name);
  int m_i;
```

```
  string m_s;

  m_s = name;
  m_i = q.pop_front();
  $display("string = %s, value = %0d", m_s, m_i);
  q.push_front(m_i);

endfunction: do_it
```

Higher Performance Version

```
function automatic void do_it(ref int q[$], ref string name);
  int m_i;
  string m s;

  m_s = name;
  m_i = q.pop_front();
  $display("string = %s, value = %0d", m_s, m_i);
  q.push front(m i);

endfunction: do it
```

```
In the lower performance version of the code, a queue of ints and a
string are copied into the function. As the queue grows in length, this
 becomes increasingly expensive. In the higher performance version,
both the int queue and the string arguments are references, this avoids
 the copy operation and speeds up the execution of the function.
```

Class Performance Guidelines

In SystemVerilog, a class encapsulates data variables and methods that operate on those variables. A class can be extended to add more variables and add to or extend the existing methods to provide new functionality. All of this convenience and functionality comes with a performance overhead which can be minimized by the following guidelines:

Avoid Unnecessary Object Construction

Constructing an object can have an overhead associated with it. As a general rule, try to minimize the number of objects created.

Lower Performance Version

```
//
// Function that returns an object handle
//
function bus_object get_next(bus_state_t bus_state);
  bus_object bus_txn = new();

  if(bus_state.status == active) begin
```

```
    bus_txn.addr = bus_state.addr;
    bus_txn.opcode = bus_state.opcode;
    bus_txn.data = bus_state.data;
    return bus_txn;
  end

  return null;

endfunction: get_next
```

Higher Performance Version

```
//
// Function that returns an object handle
//
function bus_object get_next(bus_state_t bus_state);
  bus object bus txn;

  // Only construct the bus_txn object if necessary:
  if(bus_state.status == active) begin
    bus_txn = new();
    bus_txn.addr = bus_state.addr;
    bus_txn.opcode = bus_state.opcode;
    bus_txn.data = bus_state.data;
  end

  return bus txn; // Null handle if not active

endfunction: get next
```

```
It is not necessary to construct the bus transaction object, the function will return a null handle if it is not constructed.
```

Lower Performance Version

```
task handle_bus_write;
  bus_object write_req =
     bus_object::type_id::create("write_req");

  write_bus_req_fifo.get(write_req);
  // do something with the write_req;

endtask: handle_bus_write
```

Higher Performance Version

```
task handle_bus_write;
  bus_object write_req;


  write_bus_req_fifo.get(write_req);
  // do something with the write_req;

endtask: handle_bus_write
```

Constructing the write_req object is redundant since its handle is re-assigned by the get from the bus_write_req_fifo.

Direct Variable Assignment Is Faster Than set()/get() Methods

Calling a method to update or examine a variable carries a higher overhead than direct access via the class hierarchical path.

Lower Performance Version

```
// Class with access methods
class any_thing;

int A;

function void set_A(int value);
  A = value;
endfunction: set_A

function int get_A();
  return A;
endfunction: get_A

endclass: any_thing

// code that uses this class
// and its access methods

any_thing m;
int V;

initial begin
  m = new();
  V = 1;
  repeat(10) begin
    m.set_A(V);
    V = V + m.get_A();
  end
end
```

Higher Performance Version

```
// Class with access methods
class any_thing;

int A;

function void set_A(int value);
  A = value;
endfunction: set_A

function int get_A();
  return A;
endfunction: get_A

endclass: any_thing

// code that uses this class
// and makes direct assignments

any_thing m;
int V;

initial begin
  m = new();
  V = 1;
  repeat(10) begin
    m.A = V;
    V = V + m.A;
  end
end
```

Making an assignment to the data variable within the class using its hierarchical path is more efficient than calling a method to set()/get() it. However, if the set()/get() method does more than a simple assignment - e.g. a type conversion or a checking operation on the arguments provided, then the method approach should be used.

Note that: this guideline is for performance and flouts the normal OOP guideline that data variables within a class should only be accessible via methods. Using direct access methods to get to variables improves performance, but comes at the potential cost of making the code less reusable and relies on the assumption that the user knows the name and type of the variable in question.

Avoid Method Chains

Calling a method within a class carries an overhead, nesting or chaining method calls together increases the overhead. When you implement or extend a class try to minimize the number of levels of methods involved.

Lower Performance Version

```
class mailbox_e #(type T = int); local mailbox
    #(T) mb;

    // standard mailbox API
    extern function new(int bound = 0);
    extern function int num();
    extern task put(T item);
    extern function int try_put(T item);
    extern task get(ref T item);
    extern function int try_get(ref T item);
    extern function int try_peek(ref T item);

    // extended API
    extern function void reset();
endclass : mailbox_e

function mailbox_e::new(int bound = 0); mb =
    new(bound);
endfunction

function int mailbox_e::num();
    return mb.num();
endfunction: num

task mailbox_e::put(T item);
    mb.put(item);
endtask: put

function int mailbox_e::try_put(T item);
    return mb.try_put(item);
endfunction: try_put

task mailbox_e::get(ref T item); mb.get(item);
endtask: get

function int mailbox_e::try_get(ref T item);
    return mb.try_get(item);
endfunction: try_get

function int mailbox_e::try_peek(ref T item);
    return mb.try_peek(item);
endfunction: try_peek

function void mailbox_e::reset(); T obj;
    while (mb.try_get(obj));
endfunction: reset
```

Higher Performance Version

```
class mailbox_e #(type T = integer)
  extends mailbox #(T);

extern function new(int bound = 0);
// Flushes the mailbox:
extern function void reset();

endclass: mailbox_e

function mb_e::new(int bound = 0);
  super.new(bound);
endfunction

function void mb_e::reset();
  T obj;
  while(try_get(obj));
endfunction: reset
```

The second implementation extends the mailbox directly and avoids the extra layer in the first example.

Lower Performance Version

```
class multi_method;
int i;

function void m1();
  m2();
endfunction: m1

function void m2();
  m3();
endfunction: m2

function void m3();
  i++;
endfunction: m3

endclass: multi_method
```

Higher Performance Version

```
class multi_method; int i;

function void m1();
  i++;
endfunction: m1

endclass: multi_method
```

In the first example, a function call is implemented as a chain, whereas the second example has a single method and will have a higher performance. Your code may be more complex, but it may have method call chains that you could unroll.

Array Guidelines

SystemVerilog has a number of array types which have different characteristics, it is worth considering which type of array is best suited to the task in hand. The following table summarizes the considerations.

Array Type	Characteristics	Memory Impact	Performance Impact
Static Array int a_ray[7:0];	Array size fixed at compile time. Index is by integer.	Least	Array indexing is efficient. Search of a large array has an overhead.
Dynamic Array int a_ray[];	Array size determined/changed during simulation. Index is by integer.	Less	Array indexing is efficient. Managing size is important.
Queues int a_q[$];	Use model is as FIFO/LIFO type storage Self-managed sizing Uses access methods to push and pop data	More	Efficient for ordered accesses Self managed sizing minimizes performance impact
Associative arrays int a_ray[string];	Index is by a defined type, not an integer Has methods to aid management Sized or unsized at compile time, grows with use	More	Efficient for sparse storage or random access Becomes more inefficient as it grows, but elements can be deleted Non-integer indexing can raise abstraction

For example, it may be more efficient to model a large memory space that has only sparse entries using an associative array rather than using a static array. However, if the associative array becomes large because of the number of entries then it would become more efficient to implement to use a fixed array to model the memory space.

Use Associative Array Default Values

In some applications of associative arrays there may be accesses using an index which has not been added to the array, for instance a scoreboard sparse memory or a tag of visited items. When an associative array gets an out of range access, then by default it returns an warning message together with an uninitialized value. To avoid this scenario, the array can be queried to determine if the index exists and, if not, the access does not take place. If the default variable syntax is used, then this work can be avoided with a performance improvement:

Lower Performance Version

```
// Associative array declaration - no default value:
int aa[int];

if(aa.exists(idx)) begin
  lookup = aa[idx];
end
```

Higher Performance Version

```
// Associative array declaration - setting the default to 0
int aa[int] = {default:0};

lookup = aa[idx];
```

Avoiding Work

The basic principle here is to avoid doing something unless you have to. This can manifest itself in various ways:

- Don't randomize an object unless you need to
- Don't construct an object unless you need to
- Break out of a loop once you've found what you're looking for
- Minimize the amount of string handling in a simulation - in UVM testbenches this means using the `uvm_info(), `uvm_warning(), `uvm_error(), `uvm_fatal() macros to avoid string manipulation unless the right level of verbosity has been activated

Constraint Performance Guidelines

Constrained random generation is one of the most powerful features in SystemVerilog. However, it is very easy to over constrain the stimulus generation. A little thought and planning when writing constraints can make a big difference in the performance of a testbench. Always consider the following when writing constrained random code in classes:

1. Minimize the number of active rand variables - if a value can be calculated from other random fields then it should be not be rand
2. Use minimal data types - i.e. bit instead of logic, tune vectors widths to the minimum required
3. Use hierarchical class structures to break down the randomization, use a short-circuit decision tree to minimize the work
4. Use late randomization to avoid unnecessary randomization
5. Examine repeated use of in-line constraints - it may be more efficient to extend the class
6. Avoid the use of arithmetic operators in constraints, especially *, /, % operators

7. Implication operators are bidirectional, using solve before enforces the probability distribution of the before term(s)
8. Use the pre-randomize() method to pre-set or pre-calculate state variables used during randomization
9. Use the post-randomize() method to calculate variable values that are dependent on random variables.
10. Is there an alternative way of writing a constraint that means that it is less complicated?

The best way to illustrate these points is through an example - note that some of the numbered points above are referenced as comments in the code:

Lower Performance Version:

```
class video_frame_item extends uvm_sequence_item;

typedef enum {live, freeze} live_freeze_t; // 2
typedef enum {MONO, YCbCr, RGB} video_mode_e; // 3

// Frame Packets will either be regenerated or repeated
// in the case of freeze.
rand live_freeze_t live_freeze = live; // 1

int x_pixels;
int y_pixels;
rand int length; // 1

video_mode_e mode;

rand int data_array []; // 2

// Constraints setting the data values
constraint YCbCr_inside_c                {
    foreach (data_array[i]) data_array[i] inside {[16:236]};
}
constraint RGB_inside_c                  {
    foreach (data_array[i]) data_array[i] inside {[0:255]};
}
constraint MONO_inside_c                 {
    foreach (data_array[i]) data_array[i] inside {[0:4095]};
}
// Constraints setting the size of the array
constraint YCbCr_size_c { data_array.size ==
    (2*length); // 6
}
constraint RGB_size_c { data_array.size ==
    (3*length); // 6
}
constraint MONO_size_c { data_array.size ==
    (length); // 6
}
// Frequency of live/freeze frames:
```

```
constraint live_freeze_dist_c {
    live_freeze dist { freeze := 20, live := 80};
}
// Set the frame size in pixels
constraint calc_length_c {
  length == x_pixels * y_pixels; // 6
}

// UVM Factory Registration
`uvm_object_utils(video_frame_item)

// During freeze conditions we do not want to
// randomize the data on the randomize call.
// Set the randomize mode to on/off depending on
// whether the live/freeze value.

function void pre_randomize(); // 8

  if (live_freeze == live) begin
    this.data_array.rand_mode(1);
  end
  else begin this.data_array.rand_mode(0);
  end

endfunction: pre_randomize

function void set_frame_vars(int pix_x_dim = 16,
                             int pix_y_dim = 16, video_mode_e
                                         vid_type = MONO);
  x_pixels = pix_x_dim; y_pixels
  = pix_y_dim;

  // Default constraints are off
  MONO_inside_c.constraint_mode(0);
  MONO_size_c.constraint_mode(0);
  YCbCr_inside_c.constraint_mode(0);
  YCbCr_size_c.constraint_mode(0);
  RGB_inside_c.constraint_mode(0);
  RGB_size_c.constraint_mode(0); mode =
  vid_type;

  case (vid_type)
    MONO  : begin
              this.MONO_inside_c.constraint_mode(1);
              this.MONO_size_c.constraint_mode(1);
            end
```

```
        YCbCr : begin
                        this.YCbCr_inside_c.constraint_mode(1);
                        this.YCbCr_size_c.constraint_mode(1); end
        RGB     : begin this.RGB_inside_c.constraint_mode(1);
                        this.RGB_size_c.constraint_mode(1); end
        default : `uvm_error(get_full_name(),
            "!!!!No valid video format selected!!!\n\n", UVM_LOW);
    endcase

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

endclass: video_frame_item
```

Higher Performance Version:

```
typedef enum bit {live, freeze} live_freeze_t; // 2
typedef enum bit[1:0] {MONO, YCbCr, RGB} video_mode_e; // 2
```

```
class video_frame_item extends uvm_sequence_item;

// Frame Packets will either be regenerated or repeated
// in the case of freeze.
rand live_freeze_t live_freeze = live; // 1

int length; // 1
video_mode_e mode;

bit [11:0] data_array []; // 1, 2

constraint live_freeze_dist_c {
                    live_freeze dist { freeze := 20, live := 80};
}

// UVM Factory Registration
`uvm_object_utils(video_frame_item)

function void pre_randomize(); // 8

    if (live_freeze == live) begin case(mode)
        YCbCr: begin
                        data_array = new[2*length];
                        foreach(data_array[i]) begin
```

```
                data_array[i] = $urandom_range(4095, 0);
              end
          end
      RGB: begin
             data_array = new[3*length];
             foreach(data_array[i]) begin
               data_array[i] = $urandom_range(255, 0);
             end
          end
      MONO: begin
              data_array = new[length];
              foreach(data_array[i]) begin
                data_array[i] = $urandom_range(236, 16);
              end
            end
    endcase
  end

endfunction: pre_randomize

function void set_frame_vars(int pix_x_dim = 16,
                             int pix_y_dim = 16,
                             video_mode_e  vid_type = MONO);
  length = (pix_x_dim * pix_y_dim); // 1, 6
  mode = vid_type;

endfunction: set_frame_vars

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

endclass: video_frame_item
```

The two code fragments are equivalent in functionality, but have a dramatic difference in execution time. The re-factored code makes a number of changes which speed up the generation process dramatically:

- In the original code, the size of the array is calculated by randomizing two variables - length and array size. This is not necessary since the video frame is a fixed size that can be calculated from other properties in the class.
- The length of the array is calculated using a multiplication operator inside a constraint
- In the first example, the content of the data array is calculated by the constraint solver inside a foreach() loop. This is unnecessary and is expensive for larger arrays. Since these values are within a predictable range they can be generated in the post_randomize() method.
- The enum types live_freeze_t and video_mode_e will have an underlying integer type by default, the refactored version uses the minimal bit types possible.
- The original version uses a set of constraint_mode() and rand_mode() calls to control how the randomization works, this is generally less effective than coding the constraints to take state conditions into account.
- In effect, the only randomized variable in the final example is the live_freeze bit.

Other Constraint Examples

Lower Performance Version

```
rand bit[31:0] addr;
constraint align_addr_c {
  addr%4 == 0;
}
```

Higher Performance Version

```
rand bit[31:0] addr;
constraint align_addr_c {
  addr[1:0] == 0;
}
```

The first version of the constraint uses a modulus operator to set the lowest two bits to zero, the second version does this directly avoiding an expensive arithmetic operation

Lower Performance Version

```
enum bit[3:0] {ADD, SUB, DIV, OR, AND, XOR, NAND, MULT} opcode_e;
opcode_e ins;

constraint select_opcodes_c {
  ins dist {ADD:=7, SUB:=7, DIV:=7, MULT:=7};
}
```

Higher Performance Version

```
enum bit[3:0] {ADD, SUB, DIV, OR, AND, XOR, NAND, MULT} opcode_e;
opcode_e ins;

constraint select_opcodes_c {
  ins inside {ADD, SUB, DIV, MULT};
}
```

The two versions of the constraint are equivalent in the result they produce, but the first one forces a distribution to be solved which is much more expensive than limiting the ins value to be inside a set.

Covergroup Performance Guidelines

Covergroups are basically sets of counters that are incremented when the sampled value matches the bin filter, the way to keep performance up is be as frugal as possible with the covergroup activity. The basic rules with covergroups are to manage the creation of the sample bins and the sampling of the covergroup.

Bin Control

Each coverpoint automatically translates to a set of bins or counters for each of the possible values of the variable sampled in the coverpoint. This would equate to 2**n bins where n is the number of bits in the variable, but this is typically limited by the SystemVerilog auto_bins_max variable to a maximum of 64 bins to avoid problems with naive coding (think about how many bins a coverpoint on a 32 bit int would produce otherwise). It pays to invest in covergroup design, creating bins that yield useful information will usually reduce the number of bins in use and this will help with performance. Covergroup cross product terms also have the potential to explode, but there is syntax that can be used to eliminate terms.

Lower Performance Version

```
<font size="8.33">
bit[7:0] a;
bit[7:0] b;

covergroup data_cg;
  A: coverpoint a; // 256 bins
  B: coverpoint b; // 256 bins
  A_X_B: cross A, B; // 65536 bins
endgroup: data_cg
</font>
```

Higher Performance Version

```
<font size="8.33">
covergroup data_cg;
  A: coverpoint a {
    bins zero = {0}; // 1 bin
    bins min_zone[] = {[8'h01:8'h0F]}; // 15 bins
    bins max_zone[] = {[8'hF0:8'hFE]}; // 15 bins
    bins max = {8'hFF}; // 1 bin
    bins medium_zone[16] = {[8'h10:8'hEF]}; // 16 bins
  }
  B: coverpoint b{
    bins zero = {0};
    bins min_zone[] = {[8'h01:8'h0F]};
    bins max_zone[] = {[8'hF0:8'hFE]};
    bins max = {8'hFF};
    bins medium_zone[16] = {[8'h10:8'hEF]};
  }
  A_X_B: cross A, B; // 2304 bins
endgroup: data_cg
</font>
```

In the first covergroup example, the defaults are used. Without the max_auto_bins variables in place, there would be 256 bins for both A and B and 256*256 bins for the cross and the results are difficult to interpret. With max_auto_bins set to 64 this reduces to 64 bins for A, B and the cross product, this saves on performance but makes the results even harder to understand. The right hand covergroup example creates some user bins, which reduces the number of theoretical bins down to 48 bins for A and B and 2304 for the cross. This improves performance and makes the results easier to interpret.

Sample Control

A common error with covergroup sampling is to write a covergroup that is sampled on a fixed event such as a clock edge, rather than at a time when the values sampled in the covergroup are valid. Covergroup sampling should only occur if the desired testbench behavior has occurred and at a time when the covergroup variables are a stable value. Careful attention to covergroup sampling improves the validity of the results obtained as well as improving the performance of the testbench.

Lower Performance Version

```
int data;
bit active;

covergroup data_cg @(posedge clk);
  coverpoint data iff(valid == 1) {
    bins a = {[0:4000]};
    bins b = {[10000:100000]};
    bins c = {[4001:4040]};
  }
endgroup: data_cg
```

Higher Performance Version

```
int data;
bit active;

covergroup data_cg;
  coverpoint data {
    bins a = {[0:4000]};
    bins b = {[10000:100000]};
    bins c = {[4001:4040]};
  }
endgroup: data_cg

task update_coverage;
  forever begin
    @(posedge clk);
    if(valid) begin
      data_cg.sample();
    end
  end
endtask: update_coverage
```

In the first example, the covergroup is sampled on the rising edge of the clock and the iff(valid) guard determines whether the bins in the covergroup are incremented or not, this means that the covergroup is sampled regardless of the state of the valid line. In the second example, the built-in sample() method is used to sample the covergroup ONLY when the valid flag is set. This will yield a performance improvement, especially if valid is infrequently true.

Assertion Performance Guidelines

The assertion syntax in SystemVerilog provides a very succinct and powerful way of describing temporal properties. However, this power comes with the potential to impact performance. Here are a number of key performance guidelines for writing SystemVerilog assertions, they are also good general assertion coding guidelines

Unique Triggering

The condition that starts the evaluation of a property is checked every time it is sampled. If this condition is ambiguous, then an assertion could have multiple evaluations in progress, which will potentially lead to erroneous results and will definitely place a greater load on the simulator.

Lower Performance Version

```
property req_rsp;
  @(posedge clk);
  req |=>
  (req & ~rsp)[*2]
   ##1 (req && rsp)
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

Higher Performance Version

```
property req_rsp;
  @(posedge clk);
  $rose(req) |=>
  (req & ~rsp)[*2]
   ##1 (req && rsp)
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

In the first example, the property will be triggered every time the req signal is sampled at a logic 1, this will lead to multiple triggers of the assertion. In the second example, the property is triggered on the rising edge of req which is a discrete event. Other strategies for ensuring that the triggering is unique is to pick unique events, such as states that are known to be only valid for a clock cycle.

Safety vs Liveness

A safety property is one that has a bound in time - e.g. 2 clocks after req goes high, rsp shall go high. A liveness property is not bound in time - e.g. rsp shall go high following req going high. When writing assertions it is important to consider the life-time of the check that is in progress, performance will be affected by assertions being kept in flight because there is no bound on when they complete. Most specifications should define some kind of time limit for something to happen, or there will be some kind of practical limit that can be applied to the property.

Lower Performance Version

```
property req_rsp;
  @(posedge clk);
  $(posedge req) |=>
  (req & ~rsp)[*1:2]
   ##1 (req && rsp)[->1] // Unbound condition - within any number of
clocks
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

Higher Performance Version

```
property req_rsp;
  @(posedge clk);
  $rose(req) |=>
  (req & ~rsp)[*1:4] // Bounds the condition to within 1-4 clocks
   ##1 (req && rsp)
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

Assertion Guards

Assertions can be disabled using the iff(condition) guard construct. This makes sure that the property is only sampled if the condition is true, which means that it can be disabled using a state variable. This is particularly useful for filtering assertion evaluation during reset or a time when an error is deliberately injected. Assertions can also be disabled using the system tasks $assertoff() and $asserton(), these can be called procedurally from within SystemVerilog testbench code. These features can be used to manage overall performance by de-activating assertions when they are not valid or not required.

Lower Performance Version

```
property req_rsp;
  @(posedge clk);
  $(posedge req) |=>
  (req & ~rsp)[*1:2]
   ##1 (req && rsp)[->1]
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

Higher Performance Version

```
property req_rsp;
  // Disable if reset is active:
  @(posedge clk) iff(!reset);
  $rose(req) |=>
  (req & ~rsp)[*1:4]
   ##1 (req && rsp)
   ##1 (~req && ~rsp);
endproperty: req_rsp
```

Keep Assertions Simple

A common mistake when writing assertions is to try to describe several conditions in one property. This invariably results in code that is more complex than it needs to be. Then, if the assertion fails, further debug is required to work out why it failed. Writing properties that only check one part of the protocol is easier to do, and when they fail, the reason is obvious.

Avoid Using Pass And Fail Messages

SystemVerilog assertion syntax allows the user to add pass and fail function calls. Avoid the use of pass and fail messages in your code, string processing is expensive and the simulator will automatically generate a message if an assertion fails.

Lower Performance Version

```
REQ_RSP: assert property(req_rsp) begin
  $display("Assertion REQ_RSP passed at %t", $time);
else
  $display("Error: Assertion REQ_RSP failed at %t", $time);
end
```

Higher Performance Version

```
REQ_RSP: assert property(req_rsp) ;
```

Note that even leaving a blank begin ... end for the pass clause causes a performance hit.

Avoid Multiple Clocks

SystemVerilog assertions can be clocked using multiple clocks. The rules for doing this are quite complex and the performance of a multiply clocked sequence is likely to lower than a normal sequence. Avoid using multiple clocks in an assertion, if you find yourself writing one, then it may be a sign that you are approaching the problem from the wrong angle.

UVM/Guidelines

Mentor, A Siemens Business UVM Guidelines

UVM Do's

- Define classes within packages
- Define one class per file
- Use factory registration macros
- Use message macros
- Always Specify the "name" argument to uvm_object constructor
- Manually implement do_copy(), do_compare(), etc.
- Use sequence.start(sequencer)
- Use start_item() and finish_item() for sequence items
- Use the uvm_config_db API
- Use a configuration class for each agent
- Use phase objection mechanism
- Use the phase_ready_to_end() func
- Use the run_phase() in transactors
- Use the reset/configure/main/shutdown phases in tests

UVM Don'ts

- Avoid `including a class in multiple locations
- Avoid constructor arguments other than name and parent
- Avoid field automation macros
- Avoid calling super.build_phase() from any component extended from a UVM base class
- Avoid uvm_comparer policy class
- Avoid the sequence list and default sequence
- Avoid the sequence macros (`uvm_do)
- Avoid pre_body() and post_body() in a sequence
- Avoid explicitly consuming time in sequences
- Avoid set/get_config_string/_int/_object()
- Avoid the uvm_resource_db API
- Avoid callbacks
- Avoid user defined phases
- Avoid phase jumping and phase domains (for now)
- Avoid raising and lowering objections for every transaction

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

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.

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 seqeunces
  `include "example_api_seq1.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.

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.

This is required in UVM1.2, but is always a good idea anyway.

The Field Automation macros are evil and should not be used. Calling super.build_phase() will engage the field automation functionality, even if it is not used, which will cause a performance degradation. It is OK to put your own functionality in a base class that is extended from uvm_test, uvm_env, uvm_driver, etc. If you do, calling super.build_phase() from classes extended from your base class is recommended.

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.

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.

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.

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

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

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

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

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

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()

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

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

Sequences should not have explicit delay statements (#10ns) in them. Having explicit delays reduces reuse and is illegal for doing testbench accelleration in an emulator. For more information on considerations needed when creating an emulation friendly UVM testbench, please visit the Emulation article.

Virtual Sequences

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.

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.

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.

These two features of UVM phasing are still not well understood. Avoid them for now unless you are a very advanced user.

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

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 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.

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.

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.

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

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

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.

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

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)

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

User defined plusargs are reserved by usage by the UVM committee and future expansion. For more information, see the UVM/CommandLineProcessor article.

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.

UVM/PerformanceGuidelines

Although the UVM improves verification productivity, there are certain aspects of the methodology that should be used with caution, or perhaps not at all, when it comes to performance and scalability considerations.

During the simulation run time of a UVM testbench there are two distinct periods of activity. The first is the set of UVM phases that have to do with configuring, building and connecting up the testbench component hierarchy, the second is the run-time activity where all the stimulus and analysis activity takes place. The performance considerations for both periods of activity are separate.

These performance guidelines should be read in conjunction with the other methodology cookbook guidelines, there are cases where judgement is required to trade-off performance, re-use and scalability concerns.

UVM Testbench Configuration and Build Performance Guidelines

The general requirement for the UVM testbench configuration and build process is that it should be quick so that the run time phases can get going. With small testbenches containing only a few components - e.g. up to 10, the build process should be short, however when the testbench grows in size beyond this then the overhead of using certain features of the UVM start to become apparent. With larger testbenches with 100s, or possibly 1000s of components, the build phase will be noticeably slower. The guidelines in this section apply to the full spectrum of UVM testbench size, but are most likely to give most return as the number of components increases.

Avoid auto-configuration

Auto-configuration is a methodology inherited from the OVM where a component's configuration variables are automatically set to their correct values from variables that have been set up using set_config_int(), set_config_string(), uvm_config_db #(..)::set() etc at a higher level of the component hierarchy. In order to use auto-configuration, field macros are used within a component and the super.build_phase() method needs to be called during the build_phase(), the auto-configuration process then attempts to match the fields in the component with entries in the configuration database via a method in uvm_component called apply_config_settings(). From the performance point of view, this is VERY expensive and does not scale.

Lower Performance Version

```
class my_env extends uvm_component;

bit has_axi_agent;
bit has_ahb_agent;
string system_name;

axi_agent m_axi_agent;
ahb_agent m_ahb_agent;

// Required for auto-configuration
`uvm_component_utils_begin(my_env)
  `uvm_field_int(has_axi_agent, UVM_DEFAULT)
  `uvm_field_int(has_ahb_agent, UVM_DEFAULT)
  `uvm_field_string(system_name, UVM_DEFAULT)
`uvm_component_utils_end
```

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

function void build_phase(uvm_phase phase);
  super.build_phase(phase); // Auto-configuration called here
  if(has_axi_agent == 1) begin
    m_axi_agent = axi_agent::type_id::create("m_axi_agent", this);
  end
  if(has_ahb_agent == 1) begin
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent", this);
  end
  `uvm_info("build_phase", $sformatf("%s built", system_name))
endfunction: build_phase

endclass: my_env
```

Higher Performance Version

```
class my_env extends uvm_component;

my_env_config cfg;

axi_agent m_axi_agent;
ahb_agent m_ahb_agent;

`uvm_component_utils(my_env)

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

function void build_phase(uvm_phase phase);
  // Get the configuration, note class variables not required
  if(!uvm_config_db #(my_env_config)::get(this, "", "my_env_config",
cfg)) begin
    `uvm_error("build_phase", "Unable to find my_env_config in
uvm_config_db")
  end
  if(cfg.has_axi_agent == 1) begin
    m_axi_agent = axi_agent::type_id::create("m_axi_agent", this);
  end
  if(cfg.has_ahb_agent == 1) begin
    m_ahb_agent = ahb_agent::type_id::create("m_ahb_agent", this);
  end
  `uvm_info("build_phase", $sformatf("%s built", cfg.system_name))
```

```
endfunction: build_phase

endclass: my_env
```

The recommended practice is not to use field macros in a component, and to not call super.build_phase() if the class you are extending is from a UVM component base class such as uvm_component. Even then, when a component does not have a build_phase() method implementation, the default build_phase() from the uvm_component base class will be called which will attempt to do auto-configuration. In UVM 1.1b, a fix was added that stops the apply_config_settings() method from continuing if there are no field macros in the component, this speeds up component build, but it is more efficient to avoid this method being called altogether.

Minimize the use of the uvm_config_db

The uvm_config_db is a database, as with any database it takes longer to search as it grows in size. The uvm_config_db is based on the uvm_resource and the uvm_resource_db classes. The uvm_resource_db uses regular expressions and the component hierarchy strings to make matches, it attempts to check every possible match and then returns the one that is closest to the search, this is expensive and the search time increases exponentially as the database grows. Therefore, the uvm_config_db should be used sparingly, if at all. This also applies to the set/get_config_xxx() methods since they in turn are based on the uvm_config_db.

Use configuration objects to pass configuration data to components

One way to minimize the number of uvm_config_db entries is to group component configuration variables into a configuration object. That way only one object needs to be set() in the uvm_config_db. This has reuse benefits and is the recommended way to configure reusable verification components such as agents.

Lower Performance Version

```
class static_test extends uvm_test;

// Test that builds an env containing an AXI agent

virtual axi_if AXI; // Used by the AXI agent

// Only consider the build method:
function void build_phase(uvm_phase phase);
  // Configuration code for the AXI agent
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI", AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  uvm_config_db #(virtual axi_if)::set(this, "env.axi_agent*", "v_if",
AXI);
  uvm_config_db #(uvm_active_passive_enum)::set(
                      this, "env.axi_agent*", "is_active", UVM_ACTIVE);
  uvm_config_db #(int)::set(this, "env.axi_agent*", "max_burst_size",
16);
  // Other code

endfunction: build_phase
```

```
endclass: static_test

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration parameters: virtual axi_if AXI;
uvm_active_passive_enum is_active; int
max_burst_size;

axi_driver driver; axi_sequencer
sequencer; axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI", AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  if(!uvm_config_db #(uvm_active_passive_enum)::get(this,
                                                         "", "is_active", is_active))
begin
    `uvm_error("build_phase", "is_active not found in uvm_config_db")
  end
  if(!uvm_config_db #(int)::get(
                                  this, "", "max_burst_size", max_burst_size))
begin
    `uvm_error("build_phase", "max_burst_size not found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this); if(is_active ==
  UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this); sequencer =
    axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

function void connect_phase(uvm_phase phase); monitor.AXI =
  AXI;
  if(is_active == UVM_ACTIVE) begin
    driver.AXI = AXI;
    driver.max_burst_size = max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent
```

Higher Performance Version

```
// Additional agent configuration class:
class axi_agent_config extends uvm_object;

`uvm_object_utils(axi_agent_config)

virtual axi_if AXI;
uvm_active_passive_enum is_active = UVM_ACTIVE;
int max_burst_size = 64;

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

endclass: axi_agent_config

class static_test extends uvm_test;

// Test that builds an env containing an AXI agent

axi_agent_config axi_cfg; // Used by the AXI agent

// Only consider the build method:
function void build_phase(uvm_phase phase);
  // Configuration code for the AXI agent
  axi_cfg = axi_agent_config::type_id::create("axi_cfg");
  if(!uvm_config_db #(virtual axi_if)::get(this, "", "AXI",
                                              axi_cfg.AXI)) begin
    `uvm_error("build_phase", "AXI vif not found in uvm_config_db")
  end
  axi_cfg.is_active = UVM_ACTIVE;
  axi_cfg.max_burst_size = 16;
  uvm_config_db #(axi_agent_config)::set(this,
                        "env.axi_agent*", "axi_agent_config", axi_cfg);
  // Other code endfunction:

build_phase endclass: static_test

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;
```

```
axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                 "", "axi_agent_config", cfg)) begin
    `uvm_error("build_phase", "AXI agent config object not
                                        found in uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
  if(cfg.is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

function void connect_phase(uvm_phase phase);
  monitor.AXI = cfg.AXI;
  if(cfg.is_active == UVM_ACTIVE) begin
    driver.AXI = cfg.AXI;
    driver.max_burst_size = cfg.max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent
```

The higher performance version of the example uses one uvm_config_db #(...)::set() call and two get() calls, compared with three set() and four get() calls in the lower performance version. There are also just two uvm_config_db entries compared to four. With a large number of components, this form of optimization can lead to a considerable performance boost.

Minimize the number of uvm_config_db #(...)::get() calls

The process of doing a get() from the uvm_config_db is expensive, and should only be used when really necessary. For instance, in an agent, it is only really necessary to get() the configuration object at the agent level and to assign handles to the sub-components from there. It is an unecessary overhead to have separate get() calls inside the driver and monitor components.

Lower Performance Version

```
// Agent configuration class - configured and set() by the test class
class axi_agent_config extends uvm_object;

`uvm_object_utils(axi_agent_config)

virtual axi_if AXI;
uvm_active_passive_enum is_active = UVM_ACTIVE;
int max_burst_size = 64;
```

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

endclass: axi_agent_config

// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;

axi_driver driver; axi_sequencer
sequencer; axi_monitor monitor;

function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(axi_agent_config)::get(this,
                                                                                "", "axi_agent_config", cfg)) begin
        `uvm_error("build_phase", "AXI agent config object
                                                                                          not found in uvm_config_db")
    end
    monitor = axi_monitor::type_id::create("monitor", this); if(cfg.is_active ==
    UVM_ACTIVE) begin
        driver = axi_driver::type_id::create("driver", this); sequencer =
        axi_sequencer::type_id::create("sequencer", this);
    end
endfunction: build_phase

endclass: axi_agent

// The axi monitor:
class axi_monitor extends uvm_component;

axi_if AXI; axi_agent_config
cfg;

function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(axi_agent_config)::get(this,
                                                                                  "", "axi_agent_config", cfg))
begin
        `uvm_error("build_phase", "AXI agent config object
                                                                                              not found in
uvm_config_db")
    end
    AXI = cfg.AXI;
```

```
endfunction: build_phase

endclass: axi_monitor

// The axi driver:
class axi_monitor extends uvm_component;

axi_if AXI;
int max_burst_size;
axi_agent_config cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                             "", "axi_agent_config", cfg))
begin
    `uvm_error("build_phase", "AXI agent config object
                                                      not found in
uvm_config_db")
  end
  AXI = cfg.AXI;
  max_burst_size = cfg.max_burst_size;
endfunction: build_phase

endclass: axi_driver
```

Higher Performance Version

```
// The AXI agent:
class axi_agent extends uvm_component;

// Configuration object:
axi_agent_config cfg;

axi_driver driver;
axi_sequencer sequencer;
axi_monitor monitor;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(axi_agent_config)::get(this,
                                             "", "axi_agent_config", cfg))
begin
    `uvm_error("build_phase", "AXI agent config object
                                                        not found in
uvm_config_db")
  end
  monitor = axi_monitor::type_id::create("monitor", this);
```

```
  if(cfg.is_active == UVM_ACTIVE) begin
    driver = axi_driver::type_id::create("driver", this);
    sequencer = axi_sequencer::type_id::create("sequencer", this);
  end
endfunction: build_phase

// Direct assignment to the monitor and driver variables
// from the configuration object variables.
function void connect_phase(uvm_phase phase);
  monitor.AXI = cfg.AXI;
  if(cfg.is_active == UVM_ACTIVE) begin
    driver.AXI = cfg.AXI;
    driver.max_burst_size = cfg.max_burst_size;
  end
endfunction: connect_phase

endclass: axi_agent

// The axi_monitor and axi_driver are implemented without
// a uvm_config_db #()::get()
```

The higher performance version has two fewer calls to the uvm_config_db #()::get() method, which when multiplied by a large number of components can lead to a performance improvement.

Use specific strings with the uvm_config_db set() and get() calls

The regular expression algorithm used in the search attempts to get the closest match based on the UVM component's position in the testbench hierarchy and the value of the key string. If wildcards are used in either the set() or get() process, then this adds ambiguity to the search and makes it more expensive. For instance, setting the context string to "*" means that the entire component hierarchy will be searched for uvm_config_db settings before a result is returned.

Lower Performance Version

```
// In one component, setting config for env.sb
sb_cfg = sb_config::type_id::create("sb_cfg");
// Configure content of sb_cfg ...
umv_config_db #(sb_config)::set(this, "*", "*_config", sb_cfg);

// In the env.sb component:
sb_config cfg;
if(!uvm_config_db #(sb_config)::get(this, "", "*_config", cfg)) begin
  `uvm_error(...)
end
```

Higher Performance Version

```
// In one component, setting config for env.sb
sb_cfg = sb_config::type_id::create("sb_cfg");
```

```
// Configure content of sb_cfg ...
umv_config_db #(sb_config)::set(this, "env.sb", "sb_config", sb_cfg);

// In the env.sb component:
sb_config cfg;
if(!uvm_config_db #(sb_config)::get(this, "", "sb_config", cfg)) begin
  `uvm_error(...)
end
```

In the higher performance version of this code, the scope is very specific and will only match on the single component for a single key, this cuts downs the search time in the uvm_config_db

Minimize the number of virtual interface handles passed via uvm_config_db from the TB module to the UVM environment

Consolidate the virtual interface handles into a single configuration object

An alternative to using a package to pass the virtual interface handles from the testbench top level module to the UVM test, is to create a single configuration object that contains all the virtual interface handles and to make the virtual interface assignments in the top level module before setting the configuration object in the uvm_config_db. This reduces the number of uvm_config_db entries used for passing virtual interface handles down to one.

```
// Virtual interface configuration object:
class vif_handles extends uvm_object;
`uvm_object_utils(vif_handles)

virtual axi_if AXI;
virtual ddr2_if DDR2;

endclass: vif_handles

// In the top level testbench module:
module top_tb;

import uvm_pkg::*;
import test_pkg::*;

// Instantiate the static interfaces:
axi_if AXI();
ddr2_if DDR2();

// Virtual interface handle container object:
vif_handles v_h;

// Hook up to DUT ....

// UVM initial block:
initial begin
  // Create virtual interface handle container:
```

```
  v_h = vif_handles::type_id::create("v_h");
  // Assign handles
  v_h.AXI = AXI;
  v_h.DDR2 = DDR2;
  // Set in uvm_config_db:
  uvm_config_db #(vif_handles)::set("uvm_test_top", "", "V_H", vh);
  run_test();
end

endmodule: top_tb
```

Pass configuration information through class hierarchical references

The ultimate in minimizing the use of the uvm_config_db is not to use it at all. It is perfectly possible to pass handles to configuration objects through the class hierarchy at build time. Using handle assignments is the most efficient way to do this.

Lower Performance Version

```
// In the test, with the env configuration object containing
// nested configuration objects for its agents:
function void build_phase(uvm_phase phase);
  env_cfg = env_config_object::type_id::create("env_cfg");
  // Populate the env_cfg object with axi_cfg, ddr2_cfg etc
  env = test_env::type_id::create("env");
  uvm_config_db #(env_config_object)::set(this, "env", "env_cfg",
env_cfg);
  // ...
endfunction: build_phase

// In the env, building the axi and ddr2 agents:
env_config_object cfg;

function void build_phase(uvm_phase phase);
  if(!uvm_config_db #(env_config_object)::get(this,
                                              "", "env_cfg", cfg)) begin
    `uvm_error(...)
  end
  // Create AXI agent and set configuration for it:
  axi = axi_agent::type_id::create("axi", this);
  uvm_config_db #(axi_agent_config)::set(this,
                                "axi", "axi_agent_config",
cfg.axi_cfg);
  // Also for the DDR2 agent:
  ddr2 = ddr2_agent::type_id::create("ddr2", this);
  uvm_config_db #(ddr2_agent_config)::set(this,
                             "ddr2", "ddr2_agent_config",
cfg.ddr2_cfg);
  // etc
endfunction: build_phase
```

Higher Performance Version

```
// In the test, with the env configuration object containing
// nested configuration objects for its agents:
function void build_phase(uvm_phase phase);
  env_cfg = env_config_object::type_id::create("env_cfg");
  // Populate the env_cfg object with axi_cfg, ddr2_cfg etc
  env = test_env::type_id::create("env");
  // Assign the env configuration object handle directly:
  env.cfg = env_cfg;
  // ...
endfunction: build_phase

// In the env, building the axi and ddr2 agents:
env_config_object cfg;

function void build_phase(uvm_phase phase);
  // Create AXI agent and set configuration for it:
  axi = axi_agent::type_id::create("axi", this);
  // Assign axi agents configuration handle directly:
  axi.cfg = cfg.axi_cfg;
  // Also for the DDR2 agent:
  ddr2 = ddr2_agent::type_id::create("ddr2", this);
  ddr2.cfg = cfg.ddr2_cfg;
  // etc
endfunction: build_phase
```

The higher performance example avoids the use of the uvm_config_db altogether, providing the ultimate configuration and build performance enhancement. The impact of using this approach is that it requires the assignments to be chained together; that it requires the agent code to test for a null config object handle before attempting to get the configuration object handle; and that any stimulus hierarchy needs to take care of getting handles to testbench resources such as register models.

Another major consideration with this direct approach to the assignment of configuration object handles is that if VIP is being re-used, it may well be implemented with the expectation that its configuration object will be set in the uvm_config_db. This means that there may have to be some use of the uvm_config_db to support the reuse of existing VIP.

Minimize the use of the UVM Factory

The UVM factory is there to allow UVM components or objects to be overriden with derived objects. This is a powerful technique, but whenever a component is built a lookup has to be made in a table to determine which object type to construct. If there are overrides, this lookup becomes more complicated and there is a performance penalty. Try to manage the factory overrides used to reduce the overhead of the lookup.

UVM Testbench Run-Time Performance Guidelines

The guidelines presented here represent areas of the UVM which have been seen to cause performance issues during the run-time phases of the testbench, they are mostly concerned with stimulus generation. There are other SystemVerilog coding practices that can also followed to enhance run-time performance and these are described in the SystemVerilog Performance Guidelines article.

Avoid polling the uvm_config_db for changes

Do not use the uvm_config_db to communicate between different parts of the testbench, for instance by setting a new variable value in one component and getting it inside a poll loop in another. It is far more efficient for the two components to have a handle to a common object and to reference the value of the variable within the object.

Lower Performance Version

```
// In a producer component setting the value inside a loop:
int current_id = 0;

forever begin
  // Lots of work making a transfer occur
  // Communicate the current id:
  uvm_config_db #(int)::set(null, "*", "current_id", current_id);
  current_id++;
end

// In a consumer component looking out for the current_id value
int current_id;

forever begin
   uvm_config_db #(int)::wait_modified(this, "*", "current_id");
   if(!uvm_config_db #(int)::get(this,
                              "", "current_id", current_id)) begin
     `uvm_error( ....)
   end
   // Lots of work to track down a transaction with the current_id
end
```

Higher Performance Version

```
// Config object containing current_id field:
packet_info_cfg pkt_info =
                       packet_info_cfg::type_id::create("pkt_info");
```

```
// This created in the producer component and the consumer component
// has a handle to the object:

// In the producer component:
forever begin
  // The work resulting in a current_id update
  pkt_info.current_id = current_id;
  current_id++;
end

// In the consumer component:
forever begin
  @(pkt_info.current_id);
  // Start working with the new id
end
```

The principle at work in the higher performance version is that the current_id information is inside an object. Both the consumer and the producer components share the handle to the same object, therefore when the producer object makes a change to the current_id field, it is visible to the consumer component via the handle. This avoids the use of repeated set() and get() calls in the uvm_config_db and also the use of the expensive wait_modified() method.

Do not use the UVM field macros in transactions

The UVM field macros may seem like a convenient way to ensure that the various do_copy(), do_compare() methods get implemented, but this comes at a heavy cost in terms of performance. This becomes very evident if your testbench starts to use sequence_items heavily.

Lower Performance Version

```
// APB Bus sequence_item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we;

// Field macros:
`uvm_object_utils_begin(apb_seq_item)
  `uvm_field_int(addr, UVM_DEFAULT)
  `uvm_field_int(data, UVM_DEFAULT)
  `uvm_field_enum(we, apb_opcode_e, UVM_DEFAULT)
`uvm_object_utils_end

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

endclass: apb_seq_item
```

Higher Performance Version

```
// APB Bus sequence_item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we;

`uvm_object_utils(apb_seq_item)

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

// Sequence Item convenience method prototypes:
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);
extern function void do_pack();
extern function void do_unpack();
endclass: apb_seq_item
```

Although the lower performance code example looks more compact, compiling with an -epretty flag will reveal that they expand out into many lines of code. The higher performance example shows the templates for the various uvm_object convenience methods which should be implemented manually, this will always improve performance and enhance debug should you need it.

The definitive guide on the trade-offs involved in using or not using these and the various other UVM macros can be found here.

Minimize factory overrides for stimulus objects

The UVM factory can be used to override or change the type of object that gets created when a object handle's ::type_id::create() method is called. During stimulus generation this could be applied to change the behavior of a sequence or a sequence_item without rewriting the testbench code. However, this override capability comes at a cost in terms of an extended lookup in the factory each time the object is created. To reduce the impact of creating an object overridden in the factory, create the object once and then clone it each time it is used to avoid using the factory.

Lower Performance Version

```
// apb_seq_item has been factory overridden with apb_seq_error_item
class reg_bash_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item item;
```

```
  repeat(200) begin
    item = apb_seq_item::type_id::create("item");
    start_item(item);
    assert(item.randomize() with {addr inside {[`reg_low:`reg_high]};});
    finish_item(item);
endtask:body

endclass: reg_bash_seq
```

Higher Performance Version

```
// apb_seq_item has been factory overridden with apb_seq_error_item
class reg_bash_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item original_item = apb_seq_item::type_id::create("item");
  apb_seq_item item;

  repeat(200) begin
    $cast(item, original_item.clone());
    start_item(item);
    assert(item.randomize() with {addr inside {[`reg_low:`reg_high]};});
    finish_item(item);
endtask:body

endclass: reg_bash_seq
```

The higher performance example only makes one factory create call, and uses clone() to create further copies of it, so saving the extended factory look-up each time that is expended each time round the generation loop in the lower performance example.

Avoid embedding covergroups in transactions

Embedding a covergroup in a transaction adds to its memory footprint, it also does not make sense since the transaction is disposable. The correct place to collect coverage is in a component. Transactional coverage can be collected by sampling a covergroup in a component based on transaction content.

Lower Performance Version

```
// APB Sequence item
class apb_seq_item extends uvm_sequence_item;

bit[31:0] addr;
bit[31:0] data;
apb_opcode_e we

covergroup register_space_access_cg;
```

```
ADDR_RANGE: coverpoint addr[7:0];
OPCODE: coverpoint we {
  bins rd = {APB_READ};
  bins_wr = {APB_WRITE};
}
ACCESS: cross ADDR_RANGE, OPCODE;

endgroup: register_space_access_cg;

function void sample();
  register_space_access_cg.sample();
endfunction: sample

// Rest of the sequence item ...

endclass: apb_seq_item

// Sequence producing the sequence item:
class bus_access_seq extends uvm_sequence #(apb_seq_item);

task body;
  apb_seq_item apb_item = apb_seq_item::type_id::create("apb_item");

  repeat(200) begin
    start_item(apb_item);
    assert(apb_item.randomize());
    apb_item.sample();
    finish_item(apb_item);
  end
endtask: body
```

Higher Performance Version

```
// apb_seq_item implemented without a covergroup
// Therefore bus_access_seq does not sample covergroup
// Sampling coverage in the driver:
class apb_coverage_driver extends apb_driver;

covergroup register_space_access_cg() with
                        function sample(bit[7:0] addr, apb_opcode_e we);

ADDR_RANGE: coverpoint addr;
OPCODE: coverpoint we {
  bins rd = {APB_READ};
  bins_wr = {APB_WRITE};
}
```

```
ACCESS: cross ADDR_RANGE, OPCODE;

endgroup: register_space_access_cg;

task run_phase(uvm_phase phase);
  apb_seq_item apb_item;

  forever begin
    seq_item_port.get(apb_item);
    register_space_access_cg.sample(apb_item.addr[7:0], apb_item.we);
    // Do the signal level APB cycle
    seq_item_port.item_done();
  end

endtask: run_phase

// ....
endclass: apb_coverage_driver
```

The lower performance example shows the use of a covergroup within a transaction to collect input stimulus functional coverage information. This adds a memory overhead to the transaction that is avoided by the higher performance example which collects coverage in a static component based on the content of the transaction.

Use the UVM reporting macros

The raw UVM reporting methods do not check the verbosity of the message until all of the expensive string formatting operations in the message assembly have completed. The `uvm_info(), `uvm_warning(), `uvm_error(), and `uvm_fatal() macros check the message verbosity first and then only do the string formatting if the message is to be printed.

Lower Performance Version

```
function void report_phase(uvm_phase phase);
if(errors != 0) begin
  uvm_report_error("report_phase", $sformatf(
                 %0d errors found in %0d transfers", errors, n_tfrs));
end
else if(warnings != 0) begin
  uvm_report_warning("report_phase", $sformatf(
           "%0d warnings issued for %0d transfers", warnings, n_tfrs));
end
else begin
  uvm_report_info("report_phase", $sformatf(
                               "%0d transfers with no errors", n_tfrs));
end
endfunction: report_phase
```

Higher Performance Version

```
function void report_phase(uvm_phase phase);
if(errors != 0) begin
  `uvm_error("report_phase", $sformatf(
                  "%0d errors found in %0d transfers", errors, n_tfrs))
end
else if(warnings != 0) begin
  `uvm_warning("report_phase", $sformatf(
             "%0d warnings issued for %0d transfers", warnings,
n_tfrs))
end
else begin
  `uvm_info("report_phase", $sformatf(
                     "%0d transfers with no errors", n_tfrs),
UVM_MEDIUM)
end
endfunction: report_phase
```

In the example shown, the same reports would be generated in each case, but if the verbosity settings are set to suppress the message, the higher performance version would check the verbosity before generating the strings. In a testbench where there are many potential messages and the reporting verbosity has been set to low, this can have a big impact on performance, especially if the reporting occurs frequently.

Do not use the uvm_printer class

The uvm_printer is a convenience class, originally designed to go with the use of field macros in order to print out component hierarchy or transaction content in one of several formats. The class comes with a performance overhead and its use can be avoided by using the convert2string() method for objects. The convert2string() method returns a string that can be displayed or printed using the UVM messaging macros.

Lower Performance Version

```
apb_seq_item bus_req = abp_seq_item::type_id::create("bus_req");

repeat(20) begin
  start_item(bus_req);
  assert(bus_req.randomize());
  finish_item(bus_req);
  bus_req.print();
end
```

Higher Performance Version

```
apb_seq_item bus_req = abp_seq_item::type_id::create("bus_req");

repeat(20) begin
```

```
  start_item(bus_req);
  assert(bus_req.randomize());
  finish_item(bus_req);
  `uvm_info("BUS_SEQ", bus_req.convert2string(), UVM_HIGH)
end
```

Note also that the print() method calls $display() without checking verbosity settings.

Avoid the use of get_xxx_by_name() in UVM register code

Using the get_field_by_name(), or the get_register_by_name() functions involves a regular expression search of all of the register field name or register name strings in the register model to return a handle to a field or a register. As the register model grows, this search will become more and more expensive.

Use the hierarchical path within the register model to access register content, it is far more efficient as well as being a good way to make register based stimulus reusable.

Lower Performance Version

```
task set_txen_field(bit[1:0] value);
  uvm_reg_field txen;

  txen = rm.control.get_field_by_name("TXEN");
  txen.set(value);
  rm.control.update();
endtask: set_txen_field
```

Higher Performance Version

```
task set_txen_field(bit[1:0] value);
  rm.control.txen.set(value);
  rm.control.update();
endtask: set_txen_field
```

The higher performance version of the set_txen_field avoids the expensive regular expression lookup of the field's name string.

Minimize the use of get_registers() or get_fields() in UVM register code

These calls, and others like them return return queues of object handles, this is for convenience since a queue is an unsized array. Calling these methods requires the queue to be populated which can be an overhead if the register model is a reasonable size. Repeated calls of these methods is pointless, they should only need to be called once or twice within a scope.

Lower Performance Version

```
uvm_reg regs[$];
randc int idx;
int no_regs;

repeat(200) begin
  regs = rm.encoder.get_registers();
```

```
  no_regs = regs.size();
  repeat(no_regs) begin
    tassert(this.randomize() with {idx =< no_regs;});
    assert(regs[idx].randomize());
    regs[idx].update();
  end
end
```

Higher Performance Version

```
uvm_reg regs[$];
randc int idx;
int no_regs;

regs = rm.encoder.get_registers();
repeat(200) begin
  regs.shuffle();
  foreach(regs[i]) begin
    assert(regs[i].randomize());
    regs[i].update();
  end
end
```

The higher performance version of the code only does one get_registers() call and avoids the overhead associated with the repeated call in the lower performance version.

Use UVM objections, but wisely

The purpose of raising a UVM objection is to prevent a phase from completing until a thread is ready for it to complete. Raising and dropping objections causes the component hierarchy to be traversed, with the objection being raised or dropped in all the components all the way to the top of the hierarchy. Therefore, raising and lowering an objection is expensive, becoming more expensive as the depth of the testbench hierarchy increases.

Objections should only be used by controlling threads, and the proper place to put objections is either in the run-time method of the top level test class, or in the body method of a virtual sequence. Using them in any other place is likely to be unecessary and also cause a degradation in performance.

Lower Performance Version

```
// Sequence to be called:
class adpcm_seq extends uvm_sequence #(adpcm_seq_item);
//...
task body;
  uvm_objection objection = new("objection");
  adpcm_seq_item item = adpcm_seq_item::type_id::create("item");

  repeat(10) begin
    start_item(item);
    assert(item.randomize());
    objection.raise_objection(this);
```

```
    finish_item(item);
    objection.drop_objection(this);
  end

// Inside the virtual sequence
adpcm_sequencer ADPCM;

task body;
  adpcm_seq do_adpcm = adpcm_seq::type_id::create("do_adpcm");
  do_adpcm.start(ADPCM);
endtask
```

Higher Performance Version

```
// Sequence to be called:
class adpcm_seq extends uvm_sequence #(adpcm_seq_item);
//...
task body;
  adpcm_seq_item item = adpcm_seq_item::type_id::create("item");

  repeat(10) begin
    start_item(item);
    assert(item.randomize());
    finish_item(item);
  end

// Inside the virtual sequence
adpcm_sequencer ADPCM;

task body;
  uvm_objection objection = new("objection");
  adpcm_seq do_adpcm = adpcm_seq::type_id::create("do_adpcm");
  objection.raise_objection(ADPCM);
  do_adpcm.start(ADPCM);
  objection.drop_objection(ADPCM);
endtask
```

In the higher performance version of the code, the objection is raised at the start of the sequence and dropped at the end, bracketing in time all the sequence_items sent to the driver, this is far more efficient than raising an objection per sequence_item.

Minimize the use of UVM call-backs

The implementation of call-backs in the UVM is expensive both in terms of the memory used and the code associated with registering and executing them. The complications arise mainly from the fact that the order in which the call-backs are registered is preserved. For performance, avoid the use of UVM call-backs by using alternative approaches to achieve the same functionality.

For example, register accesses can be recorded and viewed using transaction viewing either by extending the uvm_reg class or by using a call-back class. The class extended from uvm_reg overloads the pre_read() and pre_write() methods to begin a transaction when a register read() or write() method is called, and overloads the post_read() and the post_write() methods to end the transaction when the register transfer has completed. This will result in a transaction being recorded for each register access, provided the extended class is used as the base class for the register model. The alternative is to use a uvm_reg_cbs class which contains call-backs for the uvm_reg pre_read(), pre_write(), post_read() and post_write() methods. As with the extended class, the pre_xxx() methods start recording a transaction and the post_xxx() methods end recording transactions. A call back class object is then registered for each register using the package function enable_reg_recording().

Lower Performance Version Using Call Backs

```
//
// Call-Back class for recording register transactions:
//
class record reg cb extends uvm reg cbs;

  virtual      task pre write(uvm reg item rw);

  endtask

  virtual task post write(uvm reg item rw);

  endtask

  virtual task pre read(uvm reg item rw);

  endtask

  function void do record(uvm recorder recorder);

  endfunction

endclass : record reg cb

//
// Package function for enabling recording:
//
function void enable_reg_recording(uvm_reg_block reg_model,
                           reg_recording_mode_t record_mode = BY_FIELD);
  uvm_reg regs[$];
  record reg cb reg cb;
```

```
  //Set the recording mode
  uvm_config_db #(reg_recording_mode_t)::set(
                         null,"*","reg_recording_mode", record_mode);

  //Get the queue of registers
  reg_model.get_registers(regs);

  //Assign a callback object to each one
  foreach (regs[ii]) begin
    reg_cb = new({regs[ii].get_name(), "_cb"});
    uvm_reg_cb::add(regs[ii], reg_cb);
  end

  reg_cb = null;

  uvm_reg_cb::display();

endfunction : enable_reg_recording
```

Higher Performance Version Using Class Extension

```
//
// Extension of uvm_reg enables transaction recording
//
class record reg extends uvm reg;

  virtual      task pre write(uvm reg item rw);

  endtask

  virtual task post write(uvm reg item rw);

  endtask

  virtual task pre read(uvm reg item rw);

  endtask

  function void do record(uvm recorder recorder);

  endfunction

endclass : record reg
```

The main argument for using call-backs in this case is that it does not require that the register model be used with the extended class, which means that it can be 'retro-fitted' to a register model that uses the standard UVM uvm_reg class. However, this comes at the cost of a significant overhead - there is an additional call-back object for each

register in the register model and the calling of the transaction recording methods involves indirection through the UVM infrastructure to call the methods within the call-back object. Given that a register model is likely to be generated, and that there could be thousands of registers in larger designs then using the extended record_reg class will deliver higher performance with minimum inconvenience to the user.

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 logicial 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 comple 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 mathmatical 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.

Proxy

In computer programming, the proxy pattern is a software design pattern. A proxy, in its most general form, is a class functioning as an interface to something else. For the client, usage of a proxy object is similar to using the real object, because both implement the same interface.

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 coordinating 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 stimululs 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 desugn 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, coordinate the stimulus across different interfaces and the interactions between them. Usually the top level of the sequence hierarchy. AKA 'master sequence' or 'coordinator 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.

License

Copyright 2018 Mentor, A Siemens Business

For the latest product information, call us or visit **www.verificationacademy.com**

©2018 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 in this document are the trademarks of their respective owners.

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
11F, No. 120, Section 2,
Gongdao 5th Road
HsinChu City 300,
Taiwan, ROC
Phone: 886.3.513.1000
Fax: 886.3.573.4734

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Trust Tower
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004