5. Using the Register Layer Classes

5.1 Overview

The UVM *register layer classes* are used to create a high-level, object-oriented model for memory-mapped registers and memories in a design under verification (DUV). The UVM *register layer* defines several base classes that, when properly extended, abstract the read/write operations to registers and memories in a DUV. This abstraction mechanism allows the migration of verification environments and tests from block to system levels without any modifications. It also can move uniquely named fields between physical registers without requiring modifications in the verification environment or tests. Finally, UVM provides a register test sequence library containing predefined testcases you can use to verify the correct operation of registers and memories in a DUV.

A *register model* is typically composed of a hierarchy of blocks that map to the design hierarchy. Blocks can contain registers, register files and memories, as well as other blocks. The register layer classes support front-door and back-door access to provide redundant paths to the register and memory implementation, and verify the correctness of the decoding and access paths, as well as increased performance after the physical access paths have been verified. Designs with multiple physical interfaces, as well as registers, register files, and memories shared across multiple interfaces, are also supported.

Most of the UVM register layer classes must be specialized via extensions to provide an abstract view that corresponds to the actual registers and memories in a design. Due to the large number of registers in a design and the numerous small details involved in properly configuring the UVM register layer classes, this specialization is normally done by a model generator. Model generators work from a specification of the registers and memories in a design and thus are able to provide an up-to-date, correct-by-construction register model. Model generators are outside the scope of the UVM library.

Figure 21 shows how a register model is used in a verification environment.

Figure 21—Register Model in an UVM Environment

5.2 Usage Model

A register model is an instance of a register block, which may contain any number of registers, register files, memories, and other blocks. Each register file contains any number of registers and other register files. Each register contains any number of fields, which mirror the values of the corresponding elements in hardware.

Copyright © 2011 - 2015 Accellera. All rights reserved.

For each element in a register model—field, register, register file, memory or block—there is a class instance that abstracts the read and write operations on that element.

Figure 22 shows the class collaboration diagram of the register model.

Figure 22—Register Model Class Collaboration

A *block* generally corresponds to a design component with its own host processor interface(s), address decoding, and memory-mapped registers and memories. If a memory is physically implemented externally to the block, but accessed through the block as part of the block's address space, then the memory is considered as part of the block register model.

All data values are modeled as fields. *Fields* represent a contiguous set of bits. Fields are wholly contained in a register. A register may span multiple addresses. The smallest register model that can be used is a block. A block may contain one register and no memories, or thousands of registers and gigabytes of memory. Repeated structures may be modeled as register arrays, register file arrays, or block arrays.

Figure 23 shows the structure of a sample design block containing two registers, which have two and three fields respectively, an internal memory, and an external memory. Figure 24 shows the structure of the corresponding register model.

 Copyright © 2011 - 2015 Accellera. All rights reserved.

Figure 23—Design Structure of Registers, Fields, and Memories

Figure 24—Register Model Structure

Copyright © 2011 - 2015 Accellera. All rights reserved.

When using a register model, fields, registers, and memory locations are accessed through read and write methods in their corresponding abstraction class. It is the responsibility of the register model to turn these abstracted accesses into read and write cycles at the appropriate addresses via the appropriate bus driver. A register model user never needs to track the specific address or location of a field, register, or memory location, only its name.

For example, the field `ADDR` in the `CONFIG` register shown in Figure 23 can be accessed through the register model shown in Figure 24 using the `CODEC.CONFIG.ADDR.read()` method. Similarly, location 7 in the BFR memory can be accessed using the `CODEC.BFR.write(7,value)` method.

The location of fields within a physical register is somewhat arbitrary. If a field name is unique across all registers' fields within a block, it may also be accessed independently of their register location using an *alias handle* declared in the block. For example, the same `ADDR` field, being unique in name to all other fields in the `CODEC` block, may also be accessed using `CODEC.ADDR.read()`. Then, if `ADDR` is relocated from `CONFIG` to another register, any tests or environments that reference `CODEC.ADDR` will not be affected. Because a typical design has hundreds if not thousands of fields, the declaration and assignment of field aliases in a block are left as an optional feature in a register model generator.

5.2.1 Sub-register Access

When reading or writing a field using `uvm_reg_field::read()` or `uvm_reg_field::write()`, what actually happens depends on a lot of factors. If possible, only that field is read or written. Otherwise, the entire register containing that field is read or written, possibly causing unintended side effects to the other fields contained in that same register.

Consider the 128-bit register shown in Figure 25. Assuming a 32-bit data bus with a little-endian layout, accessing this entire register requires four cycles at addresses `0x00`, `0x04`, `0x08`, and `0x0C` respectively. However, field `D` can be accessed using a single cycle at address `0x01`. Since this field occupies an entire physical address, accessing it does not pose a challenge.

Figure 25—128-bit Register

Similarly, accessing field `C` can be done using a single access at address `0x00`. However, this will also access fields `B` and `A`. Accessing field `F` requires two physical accesses, at addresses `0x02` and `0x03`, but this would also access fields `E` and `G` at the same time. Accessing adjacent fields might not be an issue, but if the access has a side-effect on any of these fields, such as a `clear-on-read` field or writable field, this process will have unintended consequences.

When the underlying bus-protocol supports byte-enabling, field `C` (at address `0x00`, lane #2) can be accessed without affecting the other fields at the same address. And since field `F` is byte-aligned, it can be accessed without side effects by accessing address `0x02`, lane #3 and address `0x03`, lane #0. However, fields `B` and `A` remain inaccessible without mutual side effects as they do not individually occupy an entire byte lane.

Thus, individual field access is supported for fields that are the sole occupant of one or more byte lane(s) if the containing register does not use a user-defined front-door and the underlying bus protocol supports byte enabling. A field may also be individually-accessible if the other fields in the same byte lanes are not

 Copyright © 2011 - 2015 Accellera. All rights reserved.

affected by read or write operations. Whether a field can be individually accessible (assuming the underlying protocol supports byte-enabling) is specified by the register model generator in the `uvm_reg_field::configure()` method.

For individual field access to actually occur, two conditions must be met: the field must be identified as being the sole occupant of its byte lane by the register model generator via the `uvm_reg_field::configure()` method and the bus protocol must report that it supports byte-enables via the `uvm_reg_adapter::supports_byte_enable` property.

Finally, individual field access is only supported for front-door accesses. When using back-door accesses, the entire register—and thus all the fields it contains—will always be accessed via a peek-modify-poke operation.

5.2.2 Mirroring

The register model maintains a mirror of what it thinks the current value of registers is inside the DUT. The mirrored value is not guaranteed to be correct because the only information the register model has is the read and write accesses to those registers. If the DUT internally modifies the content of any field or register through its normal operations (e.g., by setting a status bit or incrementing an accounting counter), the mirrored value becomes outdated.

The register model takes every opportunity to update its mirrored value. For every read operation, the mirror for the read register is updated. For every write operation, the new mirror value for the written register is predicted based on the access modes of the bits in the register (`read/write`, `read-only`, `write-1-to-clear`, etc.). Resetting a register model sets the mirror to the reset value specified in the model. A mirror is not a scoreboard, however; while a mirror can accurately predict the content of registers that are not updated by the design, it cannot determine if an updated value is correct or not.

You can update the mirror value of a register to the value stored in the DUT by using the `uvm_reg_field::mirror()`, `uvm_reg::mirror()`, or `uvm_reg_block::mirror()` methods. Updating the mirror for a field also updates the mirror for all the other fields in the same register. Updating the mirror for a block updates the mirror for all fields and registers it contains. Updating a mirror in a large block may take a lot of simulation time if physical read cycles are used; whereas, updating using back-door access usually takes zero-time.

You can write to mirrored values in the register model in zero-time by using the `uvm_reg_field::set()` or `uvm_reg::set()` methods. Once a mirror value has been overwritten, it no longer reflects the value in the corresponding field or register in the DUT. You can update the DUT to match the mirror values by using the `uvm_reg::update()` or `uvm_reg_block::update()` methods. If the new mirrored value matches the old mirrored value, the register is not updated, thus saving unnecessary bus cycles. Updating a block with its mirror updates all the fields and registers the block contains with their corresponding mirror values. Updating a large block may take a lot of simulation time if physical write cycles are used; whereas, updating using back-door access usually takes zero-time. It is recommended you use this update-from-mirror process when configuring the DUT to minimize the number of write operations performed.

To access a field or register's current mirror value in zero-time, use the `uvm_reg_field::get()` or `uvm_reg::get()` methods. However, if `uvm_reg_field::set()` or `uvm_reg::set()` is used to write a desired value to the DUT, `get()` only returns the desired value, modified according to the access mode for that field or register, until the actual write to the DUT has taken place via `update()`.

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.2.3 Memories are not Mirrored

Memories can be quite large, so they are usually modeled using a sparse-array approach. Only the locations that have been written to are stored and later read back. Any unused memory location is not modeled. Mirroring a memory would require that the same technique be used.

When verifying the correct operations of a memory, it is necessary to read and write all addresses. This negates the memory-saving characteristics of a sparse-array technique, as both the memory model of the DUT and the memory would mirror, become fully populated, and duplicate the same large amount of information.

Unlike bits in fields and registers, the behavior of bits in a memory is very simple: all bits of a memory can either be written to or not. A memory mirror would then be a ROM or RAM memory model—a model that is already being used in the DUT to model the memory being mirrored. The memory mirror can then be replaced by providing back-door access to the memory model.

Therefore, using the `uvm_mem::peek()` or `uvm_mem::poke()` methods provide the exact same functionality as a memory mirror. Additionally, unlike a mirror based on observed read and write operations, using back-door accesses instead of a mirror always returns or sets the actual value of a memory location in the DUT.

5.3 Access API

Register and fields have a variety of methods to get the current value of a register or field and modify it. It is important to use the correct API to obtain the desired result.

5.3.1 read / write

The normal access API are the `read()` and `write()` methods. When using the front-door (`path=BFM`), one or more physical transactions is executed on the DUT to read or write the register. The mirrored value is then updated to reflect the expected value in the DUT register after the observed transactions.

When using the back-door (`path=BACKDOOR`), peek or poke operations are executed on the DUT to read or write the register via the back-door mechanism, bypassing the physical interface. The behavior of the registers is mimicked as much as possible to duplicate the effect of reading or writing the same value via the front-door. For example, a read from a `clear-on-read` field causes `0`'s to be poked back into the field after the peek operation. The mirrored value is then updated to reflect the actual sampled or deposited value in the register after the observed transactions.

 Copyright © 2011 - 2015 Accellera. All rights reserved.

5.3.2 peek / poke

Using the `peek()` and `poke()` methods reads or writes directly to the register respectively, which bypasses the physical interface. The mirrored value is then updated to reflect the actual sampled or deposited value in the register after the observed transactions.

5.3.3 get / set

Using the `get()` and `set()` methods reads or writes directly to the desired mirrored value respectively, without accessing the DUT. The desired value can subsequently be uploaded into the DUT using the `update()` method.

5.3.4 randomize

Using the `randomize()` method copies the randomized value in the `uvm_reg_field::value` property into the desired value of the mirror by the `post_randomize()` method. The desired value can subsequently be uploaded into the DUT using the `update()` method.

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.3.5 update

Using the `update()` method invokes the `write()` method if the desired value (previously modified using `set()` or `randomize()`) is different from the mirrored value. The mirrored value is then updated to reflect the expected value in the register after the executed transactions.

5.3.6 mirror

Using the `mirror()` method invokes the `read()` method to update the mirrored value based on the readback value. `mirror()` can also compare the readback value with the current mirrored value before updating it.

5.3.7 Concurrent Accesses

The register model can be accessed from multiple concurrent execution threads. However, it internally serializes the access to the same register to ensure predictability of the implicitly-updated mirrored value and of the other field values in the same register when a individual field is accessed.

A semaphore in each register ensures it can be read or written by only one process at a time. Any other process attempting access will block and not resume until after the current operation completes and after other processes that were blocked before it have completed their operations.

If a thread in the middle of executing a register operation is explicitly killed, it will be necessary to release the semaphore in the register it was accessing by calling the `uvm_reg::reset()` method.

 Copyright © 2011 - 2015 Accellera. All rights reserved.

5.4 Coverage Models

The UVM register library classes do not include any coverage models as a coverage model for a register will depend on the fields it contains and the layout of those fields, and a coverage model for a block will depend on the registers and memories it contains and the addresses where they are located. Since coverage model information is added to the UVM register library classes by the register model generator, that generator needs to include a suitable coverage model. Consequently, the UVM register library classes provide the necessary API for a coverage model to sample the relevant data into a coverage model.

Due to the significant memory and performance impact of including a coverage model in a large register model, the coverage model needs to handle the possibility that specific cover groups will not be instantiated or to turn off coverage measurement even if the cover groups are instantiated. Therefore, the UVM register library classes provide the necessary API to control the instantiation and sampling of various coverage models.

5.4.1 Predefined Coverage Identifiers

The UVM library has several predefined functional coverage model identifiers, as shown in Table 6. Each symbolic value specifies a different type of coverage model. The symbolic values use a one-hot encoding. Therefore, multiple coverage models may be specified by OR'ing them. Additional symbolic values may be provided for vendor-specific and user-specific coverage models that fall outside of the pre-defined coverage model types. To avoid collisions with pre-defined UVM, vendor-defined, and user-defined coverage model identifiers, bits `0` through `7` are reserved for UVM, bits `8` through `15` are reserved for vendors, and bits `16` through `23` are reserved for users. Finally, bits `24` and above are reserved for future assignment.

Table 6—Pre-defined Functional Coverage Type Identifiers

Identifier	Description
UVM_NO_COVERAGE	No coverage models.
UVM_CVR_REG_BITS	Coverage models for the bits read or written in registers.
UVM_CVR_ADDR_MAP	Coverage models for the addresses read or written in an address map.
UVM_CVR_FIELD_VALS	Coverage models for the values of fields.
UVM_CVR_ALL	All coverage models.

5.4.2 Controlling Coverage Model Construction and Sampling

By default, coverage models are not included in a register model when it is instantiated. To be included, they must be enabled via the `uvm_reg::include_coverage()` method. It is recommended register-level coverage models are only included in unit-level environments; block-level coverage models may be included in block and system-level environments.

```
uvm_reg::include_coverage("*", UVM_CVR_REG_BITS + UVM_CVR_FIELD_VALS);
```

Furthermore, the sampling for a coverage model is implicitly disabled by default. To turn the sampling for specific coverage models on or off, use the `uvm_reg_block::set_coverage()`, `uvm_reg::set_coverage()`, and `uvm_mem::set_coverage()` methods.

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.5 Constructing a Register Model

This section describes how to construct a UVM register model to represent different register and memory access and composition structures. The target audience for this section is generator writers. End users of the register model need not be familiar with the model construction process, only the final structure of the model.

5.5.1 Field Types

There is usually no need to construct field types. Fields are simple instantiations of the `uvm_reg_field` class. A field type may only be needed to specify field-level constraints, which could also be specified in the containing register.

A field type is constructed using a class extended from the `uvm_reg_field` class. There must be one class per unique field type. The name of the field type is created by the register model generator. The name of the field type class must be unique within the scope of its declaration. The field type class must include an appropriate invocation of the `` `uvm_object_utils() `` macro.

```
class my_fld_type extends uvm_reg_field;
   `uvm_object_utils(my_fld_type)
endclass
```

Field types are instantiated in the `build()` method of the containing register types.

5.5.1.1 Class Properties and Constraints

A separate constraint block should be defined for each aspect being constrained—e.g., one to keep it valid, one to keep it reasonable—so they can be turned off individually. The name of a constraint block shall be indicative of its purpose. Constraints shall constrain the `value` class property. Additional state variables may be added to the field class if they facilitate the constraints.

```
class my_fld_type extends uvm_reg_field;
   constraint valid {
      value inside {0, 1, 2, 4, 8, 16, 32};
   };
endclass
```

To ensure state variable and constraint block names do not collide with other symbols in `uvm_reg_field` base class, it is recommended their names be in all `UPPERCASE`.

If the `post_randomize()` method is overridden, it must call `super.post_randomize()`.

5.5.1.2 Constructor

The constructor must be a valid `uvm_object` constructor. The constructor shall call the `uvm_reg_field::new()` method with appropriate argument values for the field type.

```
class my_fld_type extends uvm_reg_field;

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

   endclass
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

5.5.1.3 Predefined Field Access Policies

The access policy of a field is specified using the `uvm_reg_field::configure()` method, called from the `build()` method of the register that instantiates it.

Table 7 shows the pre-defined access policies for `uvm_reg_field`. Unless otherwise stated, the effect of a read cycle on the current value is performed after the current value has been sampled for read-back. Additional field access policies may be defined using the `uvm_reg_field::define_access()` method and by modeling their behavior by extending the `uvm_reg_field` or `uvm_reg_cbs` classes.

Table 7—Pre-defined Field Access Policies

Access Policy	Description	Effect of a Write on Current Field Value	Effect of a Read on Current Field Value	Read-back Value
RO	Read Only	No effect.	No effect.	Current value
RW	Read, Write	Changed to written value.	No effect.	Current value
RC	Read Clears All	No effect.	Sets all bits to `0`'s.	Current value
RS	Read Sets All	No effect.	Sets all bits to `1`'s.	Current value
WRC	Write, Read Clears All	Changed to written value.	Sets all bits to `0`'s.	Current value
WRS	Write, Read Sets All	Changed to written value.	Sets all bits to `1`'s.	Current value
WC	Write Clears All	Sets all bits to `0`'s.	No effect.	Current value
WS	Write Sets All	Sets all bits to `1`'s.	No effect.	Current value
WSRC	Write Sets All, Read Clears All	Sets all bits to `1`'s.	Sets all bits to `0`'s.	Current value
WCRS	Write Clears All, Read Sets All	Sets all bits to `0`'s.	Sets all bits to `1`'s.	Current value
W1C	Write 1 to Clear	If the bit in the written value is a `1`, the corresponding bit in the field is set to `0`. Otherwise, the field bit is not affected.	No effect.	Current value
W1S	Write 1 to Set	If the bit in the written value is a `1`, the corresponding bit in the field is set to `1`. Otherwise, the field bit is not affected.	No effect.	Current value
W1T	Write 1 to Toggle	If the bit in the written value is a `1`, the corresponding bit in the field is inverted. Otherwise, the field bit is not affected.	No effect.	Current value
W0C	Write 0 to Clear	If the bit in the written value is a `0`, the corresponding bit in the field is set to `0`. Otherwise, the field bit is not affected.	No effect.	Current value

Copyright © 2011 - 2015 Accellera. All rights reserved.

Table 7—Pre-defined Field Access Policies (Continued)

Access Policy	Description	Effect of a Write on Current Field Value	Effect of a Read on Current Field Value	Read-back Value
W0S	Write 0 to Set	If the bit in the written value is a `0`, the corresponding bit in the field is set to `1`. Otherwise, the field bit is not affected.	No effect.	Current value
W0T	Write 0 to Toggle	If the bit in the written value is a `0`, the corresponding bit in the field is inverted. Otherwise, the field bit is not affected.	No effect.	Current value
W1SRC	Write 1 to Set, Read Clears All	If the bit in the written value is a `1`, the corresponding bit in the field is set to `1`. Otherwise, the field bit is not affected.	Sets all bits to `0`'s.	Current value
W1CRS	Write 1 to Clear, Read Sets All	If the bit in the written value is a `1`, the corresponding bit in the field is set to `0`. Otherwise, the field bit is not affected.	Sets all bits to `1`'s.	Current value
W0SRC	Write 0 to Set, Read Clears All	If the bit in the written value is a `0`, the corresponding bit in the field is set to `1`. Otherwise, the field bit is not affected.	Sets all bits to `0`'s.	Current value
W0CRS	Write 0 to Clear, Read Sets All	If the bit in the written value is a `0`, the corresponding bit in the field is set to `0`. Otherwise, the field bit is not affected.	Sets all bits to `1`'s.	Current value
WO	Write Only	Changed to written value.	No effect.	Undefined
WOC	Write Only Clears All	Sets all bits to `0`'s.	No effect.	Undefined
WOS	Write Only Sets All	Sets all bits to `1`'s.	No effect.	Undefined
W1	Write Once	Changed to written value if this is the first write operation after a hard reset. Otherwise, has no effect.	No effect.	Current value
WO1	Write Only, Once	Changed to written value if this is the first write operation after a hard reset. Otherwise, has no effect.	No effect.	Undefined

5.5.1.4 IP-XACT Field Access Mapping

Table 8 — Table 12 show the mapping between the IEEE 1685-2009 field access policies (as specified by the `fieldData` group, see section 6.10.9.2 of the IEEE 1685-2009 Standard) and the pre-defined access policies for `uvm_reg_field`. Several combinations of `access`, `modifiedWriteValue`, and `readAction` are specified as `n/a` because they do not make practical sense. However, they can always be modeled as user-defined fields if they are used in a design.

 Copyright © 2011 - 2015 Accellera. All rights reserved.

Table 8—IP-XACT Mapping for access==read-write

access == read-write				
modifiedWriteValue	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	RW	WRC	WRS	User-defined
oneToClear	W1C	n/a	W1CRS	User-defined
oneToSet	W1S	W1SRC	n/a	User-defined
oneToToggle	W1T	n/a	n/a	User-defined
zeroToClear	W0C	n/a	W0CRS	User-defined
zeroToSet	W0S	W0SRC	n/a	User-defined
zeroToToggle	W0T	n/a	n/a	User-defined
clear	WC	n/a	WCRS	User-defined
set	WS	WSRC	n/a	User-defined
modify	User-defined	User-defined	User-defined	User-defined

Table 9—IP-XACT Mapping for access==read-only

access == read-only				
modifiedWriteValue	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	RO	RC	RS	User-defined
All others	n/a	n/a	n/a	n/a

Table 10—IP-XACT Mapping for access==write-only

access == write-only				
modifiedWriteValue	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	WO	n/a	n/a	n/a
clear	WOC	n/a	n/a	n/a
set	WOS	n/a	n/a	n/a
All others	n/a	n/a	n/a	n/a

Copyright © 2011 - 2015 Accellera. All rights reserved.

Table 11—IP-XACT Mapping for access==read-writeOnce

access == read-writeOnce				
modifiedWriteValue	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	W1	n/a	n/a	n/a
All others	n/a	n/a	n/a	n/a

Table 12—IP-XACT Mapping for access==writeOnce

access == writeOnce				
modifiedWriteValue	**readAction**			
	Unspecified	**clear**	**set**	**modify**
Unspecified	WO1	n/a	n/a	n/a
All others	n/a	n/a	n/a	n/a

5.5.1.5 Reserved Fields

There is no pre-defined field access policy for reserved fields. That is because "reserved" is a documentation concept, not a behavioral specification. Reserved fields should be left unmodelled (where they will be assumed to be RO fields filled with 0's), modeled using an access policy that corresponds to their actual hardware behavior or not be compared using `uvm_reg_field::set_compare(0)`.

If reserved fields are not to be used, they should be identified with the `NO_REG_TESTS` attribute and have a user-defined behavior extension that will issue an error message if they are used.

5.5.1.6 User-defined Field Access Policy

The UVM field abstraction class contains several predefined field access modes. The access modes are used, in combination with observed read and write operations, to determine the expected value of a field. Although most fields fall within one of the predefined access policies, it is possible to design a field that behaves predictably but differently from the predefined ones.

New access policy identifiers can similarly be defined to document the user-defined behavior of the field.

```
class protected_field extends uvm_reg_field;
   local static bit m_protected = define_access("Protected");
   ...
endclass
```

The behavior of the user-defined field access policy is implemented by extending the pre/post read/write virtual methods in the field abstraction class or in the field callback methods. For example, a protected field that can only be written if another field is set to a specific value can be modeled as shown below:

```
class protected_field extends uvm_reg_field;
   local uvm_reg_field protect_mode;
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   virtual task pre_write(uvm_reg_item rw);
      // Prevent the write if protect mode is ON
      if (protect_mode.get()) begin
         rw.value = value;
   endtask
endclass
```

You can modify the behavior of any field to the user-specified behavior by registering a callback extension with that field. First, define the callback class; then, register an instance of it with an instance of the field whose behavior you want to modify:

```
class protected_field_cb extends uvm_reg_cbs;
   local uvm_reg_field protect_mode;
   virtual task pre_write(uvm_reg_item rw);
      // Prevent the write if protect mode is ON
      if (protect_mode.get()) begin
         uvm_reg_field field;
         if ($cast(field,rw.element))
            rw.value = field.get();
      end
   endtask
endclass

protected_field_cb protect_cb = new("protect_cb",protect_mode)
uvm_callbacks#(my_field_t, uvm_reg_cbs)::add(my_field, protect_cb);
```

5.5.1.7 Field Usage vs. Field Behavior

The access mode of a field is used to specify the physical behavior of the field so the mirror can track, as best as it can, the value of the field. It is not designed to specify whether or not it is suitable or functionally correct to use the field in a particular fashion from the application's perspective.

For example, a configuration field could be designed to be written only once by the software after the design comes out of reset. If the design does not support dynamic reprovisioning, it may not be proper to subsequently modify the value of that configuration field. Whether the field should be specified as write-once (`W1`) depends on the hardware functionality. If the hardware does not prevent the subsequent write operation, the field should be specified as read-write as that would accurately reflect the actual behavior of the field.

To include usage assertions to specific fields (e.g., specifying a configuration field is never written to more than once despite the fact that doing so is physically possible), implement that check in an extension of the field abstraction class, but do not prevent the mirror from reflecting that value in the hardware.

```
class config_once_field extends uvm_reg_field;
   local bit m_written = 0;
   virtual task pre_write(uvm_reg_item rw);
      if (m_written)
         `uvm_error(field.get_full_name(), "...");
      m_written = 1;
   endtask: pre_write

   virtual function reset(string kind = "HARD");
      if (has_reset(kind))
      m_written = 0;
      super.reset(kind);
   endfunction
endclass
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.5.2 Register Types

A register type is constructed using a class extended from the `uvm_reg` class. There must be one class per unique register type. The name of the register type is created by the register model generator. The name of the register type class must be unique within the scope of its declaration. The register type class must include an appropriate invocation of the `` `uvm_object_utils() `` macro.

```
class my_reg_type extends uvm_reg;
   `uvm_object_utils(my_reg_type)
endclass
```

Register types are instantiated in the `build()` method of the block and register file types.

5.5.2.1 Class Properties and Constraints

The register type must contain a public class property for each field it contains. The name of the field class property shall be the name of the field. The field class property shall have the `rand` attribute. Field class properties may be arrays.

```
class my_reg_type extends uvm_reg;
   rand uvm_reg_field F1;
   rand uvm_reg_field F2[3];
endclass
```

To ensure field names do not collide with other symbols in the `uvm_reg` base class, it is recommended their names be in all `UPPERCASE`.

Constraints, if any, should be defined in separate blocks for each aspect being constrained. This allows them to be turned off individually. The name of a constraint block shall be indicative of its purpose. Constraints shall constrain the `value` class property of each field in the register. Additional state variables may be added to the register type class if they facilitate the constraints. If the `post_randomize()` method is overridden, it must call the `super.post_randomize()` method.

If a register has only one field, then you would not want to have to write:

```
R.randomize() with (value.value == 5);
```

Instead, instantiate a private dummy field and include a rand class property named `value` in the register class. A constraint shall keep the `value` class property equal to the field's `value` class property.

```
class my_reg_type extends uvm_reg;
   rand uvm_reg_data_t value;
   local rand uvm_reg_field _dummy;

   constraint _dummy_is_reg {
      _dummy.value == value;
   }
endclass
```

Then, randomizing an instance of the register looks like the more natural:

```
R.randomize() with (value == 5);
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

5.5.2.2 Constructor

The constructor must be a valid `uvm_object` constructor. The constructor shall call the `uvm_reg::new()` method with appropriate argument values for the register type.

```
class my_reg_type extends uvm_reg;
   function new(string name = "my_reg_type");
      super.new(.name(name),
                .n_bits(32),
                .has_coverage(UVM_NO_COVERAGE));
   endfunction
endclass
```

5.5.2.3 Build() Method

A virtual `build()` function, with no arguments, shall be implemented.

The `build()` method shall instantiate all field class properties using the class factory. Because the register model is a `uvm_object` hierarchy, not a `uvm_component` hierarchy, no parent reference is specified and the full hierarchical name of the register type instance is specified as the context. The `build()` method shall call the `uvm_reg_field::configure()` method for all field class properties with the appropriate argument values for the field instance and specifying `this` as the field parent.

```
class my_reg_type extends uvm_reg;
   virtual function build();
      this.F1 = uvm_reg_field::type_id::create(.name("F1"),
                                               .parent(null),
                                               .contxt(get_full_name()));
      this.F1.configure(this, ...);

   endfunction
endclass
```

5.5.2.4 Additional Methods

Register model generators are free to add access methods to abstract common operations, For example, a read-modify-write method could be added to a register model:

```
class my_reg extends uvm_reg;
   ...
   task RMW(output uvm_status_e    status;
            input  uvm_reg_data_t  data;
            input  uvm_reg_data_t  mask; ...);
      uvm_reg_data_t tmp;
      read(status, tmp, ...);
      tmp &= ~mask;
      tmp |= data & mask;
      write(status, tmp, ...);
  endtask
endclass
```

Although allowed by UVM, such additional methods are not part of the standard. To avoid collisions with class members that may be added in the future, the name of these methods should be in `UPPERCASE` or be given a generator-specific prefix.

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.5.2.5 Coverage Model

A register-level coverage model is defined and instantiated in the register type class. It measures the coverage of read and write operations and field values on each instance of that register type. The `uvm_reg::sample()` or `uvm_reg::sample_values()` methods shall be used to trigger the sampling of a coverage point based on the data provided as argument or data gathered from the current state of the register type instance. The sampling of the coverage model shall only occur if sampling for the corresponding coverage model has been turned on, as reported by the `uvm_reg::get_coverage()` method.

```
class my_reg extends uvm_reg;
   protected uvm_reg_data_t m_current;
   protected uvm_reg_data_t m_data;
   protected bit            m_is_read;

   covergroup cg1;
      ...
   endgroup
   ...
   virtual function void sample(uvm_reg_data_t data,
                                uvm_reg_data_r byte_en,
                                bit            is_read,
                                uvm_reg_map    map);
      if (get_coverage(UVM_CVR_REG_BITS)) begin
         m_current = get();
         m_data    = data;
         m_is_read = is_read;
         cg1.sample();
      endif
   endfunction
endclass
```

All the coverage models that may be included in the register type shall be reported to the `uvm_reg` base class using the `uvm_reg::build_coverage()` method when `super.new()` is called or the `uvm_reg::add_coverage()` method. If no functional coverage models are included in the generated register type, `UVM_NO_COVERAGE` shall be specified. Register-level coverage groups shall only be instantiated in the constructor if the construction of the corresponding coverage model is enabled, as reported by the `uvm_reg::has_coverage()` method.

```
class my_reg_typ extends uvm_reg;
   ...
   covergroup cg1;
      ...
   endgroup

   covergroup cg_vendor;
      ...
   endgroup

   function new(string name = "my_reg_typ");
      super.new(name, 32, build_coverage(UVM_CVR_REG_BITS + VENDOR_CVR_REG));
      if (has_coverage(UVM_CVR_REG_BITS))
         cg1 = new();
      if (has_coverage(VENDOR_CVR_REG))
         cg_vendor = new();
   endfunction
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   ...
endclass
```

The content, structure, and options of the coverage group is defined by the register model generator and is outside the scope of UVM.

5.5.3 Register File Types

A register file type is constructed using a class extended from the `uvm_reg_file` class. There must be one class per unique register file type. The name of the register file type is created by the register model generator. The name of the register file type class must be unique within the scope of its declaration. The register file type class must include an appropriate invocation of the `` `uvm_object_utils() `` macro.

```
class my_rf_type extends uvm_reg_file;
   `uvm_object_utils(my_rf_type)
endclass
```

Register file types are instantiated in the `build()` method of the block and register file types.

5.5.3.1 Class Properties

A register file type must contain a public class property for each register it contains. The name of the register class property shall be the name of the register. The type of the register class property shall be the name of the register type. Each register class property shall have the `rand` attribute. Register class properties may be arrays.

```
class my_rf_type extends uvm_reg_file;
   rand my_reg1_type R1;
   rand my_reg2_type R2[3];
endclass
```

Register files can contain other register files. A register file type must contain a public class property for each register file it contains. The name of the register file class property shall be the name of the register file. The type of the register file class property shall be the name of the register file type. The register file class property shall have the `rand` attribute. Register file class properties may be arrays.

```
class my_rf_type extends uvm_reg_file;
   rand my_regfile1_type RF1;
   rand my_regfile2_type RF2[3];
endclass
```

To ensure register and register file names do not collide with other symbols in the register file abstraction base class, it is recommended their name be in all `UPPERCASE` or prefixed with an underscore (_).

The register file type may contain a constraint block for each cross-register constraint group it contains. The name of the constraint block shall be indicative of its purpose. Constraints shall constraint the `uvm_reg_field::`*`value`* class property of the fields in the registers contained in the register file. Additional state variables may be added to the register field type if they facilitate the constraints.

5.5.3.2 Constructor

The constructor must be a valid `uvm_object` constructor. The constructor shall call the `uvm_reg::configure()` method with appropriate argument values for the register type.

Copyright © 2011 - 2015 Accellera. All rights reserved.

```
class my_rf_type extends uvm_reg_file;
   function new(string name = "my_rf_type");
      super.(name);
   endfunction
endclass
```

5.5.3.3 build() Method

A virtual `build()` function, with no arguments, shall be implemented.

The `build()` method shall instantiate all register and register file class properties using the class factory. The name of the register or register file instance shall be prefixed with the name of the enclosing register file instance. Because the register model is a `uvm_object` hierarchy, not a `uvm_component` hierarchy, no parent reference is specified and the full hierarchical name of the block parent of the register file type instance is specified as the context. The `build()` method shall call the `configure()` method for all register and register file class properties, specifying `get_block()` for the parent block and `this` for the parent register file. The `build()` method shall call the `build()` method for all register and register file class properties.

```
class my_rf_type extends uvm_reg_file;
   virtual function build();
      uvm_reg_block blk = get_block();

      this.RF1 = my_rf1_type::type_id::create(
                              .name($psprintf("%s.rf1", get_name())),
                              .parent(null),
                              .contxt(blk.get_full_name()));
      this.RF1.configure(get_block(), this, ...);
      this.RF1.build();
      this.RF1.add_hdl_path();
   endfunction
endclass
```

5.5.3.4 map() Method

A virtual `map()` function, with `uvm_reg_map` and address offset arguments, shall be implemented. The `map()` method shall call `uvm_reg_map::add_reg()` for all register class properties, adding the value of the address offset argument to the offset of the register in the register file. The `map()` method shall call the `map()` method of all register file class properties, adding the value of the address offset argument to the offset of the register file base offset. The `map()` method may call the `add_hdl_path()` method for all register or register file class properties with appropriate argument values for the register or register file instance.

```
class my_rf_type extends uvm_reg_file;
   virtual function map(uvm_reg_map mp, uvm_reg_addr_t offset);
      mp.add_reg(this.R1, offset + 'h04, ...);
      mp.add_reg(this.R2, offset + 'h08, ...);
      this.RF1.map(mp, offset + 'h200);
   endfunction
endclass
```

5.5.3.5 set_offset() Method

A virtual `set_offset()` function, with a `uvm_reg_map` and address offset arguments, may also be implemented. The `set_offset()` method shall call the `set_offset()` method for all register and

 Copyright © 2011 - 2015 Accellera. All rights reserved.

register file class properties with appropriate argument values for the each instance, adding the value of the address offset argument to the offset of the register and register file base offset.

```
class my_rf_type extends uvm_reg_file;
   virtual function set_offset(uvm_reg_map mp, uvm_reg_addr_t offset);
      this.R1.set_offset(mp, offset + 'h04, ...);
      this.R2.set_offset(mp, offset + 'h08, ...);
      this.RF1.set_offset(mp, offset + 'h200);
   endfunction
endclass
```

5.5.4 Memory Types

A memory type is constructed using a class extended from the `uvm_mem` class. There must be one class per unique memory type. The name of the memory type is created by the register model generator. The name of the memory type class must be unique within the scope of its declaration. The memory type class must include an appropriate invocation of the `` `uvm_object_utils() `` macro.

```
class my_mem_type extends uvm_mem;
   `uvm_object_utils(my_mem_type)
endclass
```

Memory types are instantiated in the `build()` method of the block and register file types.

5.5.4.1 Class Properties

The memory type need not contain any class property.

5.5.4.2 Constructor

The constructor must be a valid `uvm_object` constructor. The constructor shall call the `uvm_mem::new()` method with appropriate argument values for the memory type.

```
class my_mem_type extends uvm_mem;
   function new(string name = "my_mem_type");
      super.new(name, ...);
   endfunction
endclass
```

5.5.4.3 Coverage Model

A memory-level coverage model is defined and instantiated in the memory type class. It measures the coverage of the accessed offsets on each instance of that memory type. The `uvm_mem::sample()` method shall be used to trigger the sampling of a coverage point, based on the data provided as an argument or gathered from the current state of the memory type instance. The sampling of the coverage model shall only occur if sampling for the corresponding coverage model has been turned on, as reported by the `uvm_mem::get_coverage(()` method.

```
class my_mem extends uvm_mem;
   local uvm_reg_addr_t m_offset;

   covergroup cg_addr;
      ...
   endgroup
   ...
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   virtual function void sample(uvm_reg_addr_t offset,
                                bit            is_read,
                                uvm_reg_map    map);
      if (get_coveragen(UVM_CVR_ADDR_MAP)) begin
         m_offset  = offset;
         cg_addr.sample();
      end
   endfunction
endclass
```

All the coverage models that may be included in the memory type shall be reported to the `uvm_mem` base class using `uvm_mem::build_coverage()` when `super.new()` is called or using the `uvm_mem::add_coverage()` method. If no functional coverage models are included in the generated memory type, `UVM_NO_COVERAGE` shall be specified. Memory-level coverage groups shall only be instantiated in the constructor if the construction of the corresponding coverage model is enabled, as reported by the `uvm_mem::has_coverage()` method.

```
class my_mem extends uvm_mem;
   ...
   covergroup cg_addr;
      ...
   endgroup

   function new(string name = "my_mem");
      super.new(name, ..., build_coverage(UVM_CVR_ADDR_MAP));
      if (has_coverage(UVM_CVR_ADDR_MAP))
         cg_addr = new();
   endfunction: new
   ...
endclass : my_mem
```

The content, structure, and options of the coverage group is defined by the register model generator and is outside the scope of UVM.

5.5.5 Block Types

A block type is constructed using a class extended from the `uvm_reg_block` class. There must be one class per unique block type. The name of the block type is created by the register model generator. The name of the block type class must be unique within the scope of its declaration. The block type class must include an appropriate invocation of the `` `uvm_object_utils() `` macro.

```
class my_blk_type extends uvm_reg_block;
   `uvm_object_utils(my_blk_type)
endclass
```

Block types are instantiated in the `build()` method of other block types and in verification environments.

5.5.5.1 Class Properties

The block type must contain a class property for each named address map it contains. The name of the address map class property shall be the name of the address map. The type of the address map class property shall be `uvm_reg_map`. The address map class property shall not have the `rand` attribute. Address map class properties shall not be arrays.

```
class my_blk_type extends uvm_reg_block;
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   uvm_reg_map AHB;
   uvm_reg_map WSH;
endclass
```

The block type must contain a class property for each register it contains. The name of the register class property shall be the name of the register. The type of the register class property shall be the name of the register type. The register class property shall have the `rand` attribute. Register class properties may be arrays.

```
class my_blk_type extends uvm_reg_block;
   rand my_r1_type R1;
   rand my_r2_type R2[3];
endclass
```

The block type must contain a class property for each register file it contains. The name of the register file class property shall be the name of the register file. The type of the register file class property shall be the name of the register file type. The register file class property shall have the `rand` attribute. Register file class properties may be arrays.

```
class my_blk_type extends uvm_reg_block;
   rand my_rf1_type RF1;
   rand my_rf2_type RF2[3];
endclass
```

The block type must contain a class property for each memory it contains. The name of the memory class property shall be the name of the memory. The type of the memory class property shall be the name of the memory type. The memory class property should not have the `rand` attribute. Memory class properties may be arrays.

```
class my_blk_type extends uvm_reg_block;
   my_mem1_type RAM1;
   my_mem2_type RAM2[3];
endclass
```

The block type must contain a class property for each sub-block it contains. The name of the sub-block class property shall be the name of the sub-block. The type of the sub-block class property shall be the name of the sub-block type. The sub-block class property shall have the `rand` attribute. Sub-block class properties may be arrays.

```
class my_blk_type extends uvm_reg_block;
    rand my_blk1_type BLK1;
    rand my_blk2_type BLK2[3];
endclass
```

To ensure register, register file, memory and block names do not collide with other symbols in `uvm_reg_block` base class, it is recommended their name be in all `UPPERCASE` or prefixed with an underscore (_).

Constraints, if any, should be defined in separate blocks for each aspect being constrained. This allows them to be turned off individually. The name of a constraint block shall be indicative of its purpose. Constraints shall constraint the `uvm_reg_field::`*`value`* class property of the fields in lower-level registers. Additional state variables may be added to the block type class if they facilitate the constraints. If the `post_randomize()` method is overridden, it must call the `super.post_randomize()` method.

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.5.5.2 Constructor

The constructor must be a valid `uvm_object` constructor. The constructor shall call the `uvm_reg_block::new()` method with appropriate argument values for the block type.

```
class my_blk_type extends uvm_reg_block;
   function new(string name = "my_blk_type");
      super.new(.name(name), .has_coverage(UVM_NO_COVERAGE));
   endfunction
endclass
```

5.5.5.3 build() Method

A virtual `build()` function, with no arguments, shall be implemented.

The `build()` method shall instantiate all named address maps by calling the `uvm_reg_block::create_map()` method, specifying appropriate argument values for the address map in the block type. One of the named address maps shall be assigned to the `uvm_reg_block::default_map` class property.

```
class my_blk_type extends uvm_reg_block;
   virtual function build();
         this.AHB = create_map();
         this.WSH = create_map();
         this.default_map = this.AHB;
   endfunction
endclass
```

If the block does not contain any named address maps, the `build()` method shall instantiate an anonymous address map by calling the `uvm_reg_block::create_map()` method, specifying the name of the address map and other appropriate argument values for the block type, and assign those to the `uvm_reg_block::default_map` class property.

```
class my_blk_type extends uvm_reg_block;
   virtual function build();
      this.default_map = create_map(.name("default_map",
                                    .base_addr(`h1000),
                                    .n_bytes(4),
                                    .endian(UVM_LITTLE_ENDIAN));
   endfunction
endclass
```

The `build()` method shall instantiate all register, register file, memory, and sub-block class properties using the class factory. Because the register model is a `uvm_object` hierarchy, not a `uvm_component` hierarchy, no parent reference is specified and the full hierarchical name of the block type instance is specified as the context. The `build()` method shall call the `configure()` method for all register, register file, memory, and sub-block class properties, specifying `this` as the block parent and `null` as the register file parent. The `build()` method shall call the `build()` method for all register, register file, and sub-block class properties. The `build()` method may call the `add_hdl_path()` method for any register, register file, memory, or sub-block class properties with the appropriate argument values for the register, register file, memory, or sub-block instance.

```
class my_blk_type extends uvm_reg_block;
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   virtual function build();
      this.BLK1 = my_blk1_type::type_id::create("BLK1", null,
                                                get_full_name());
      this.BLK1.configure(this, ...);
      this.BLK1.build();
   endfunction
endclass
```

After a register or memory has been created, the `build()` method shall call the appropriate `uvm_reg_map::add_*()` method for all address maps where the register, register file, or memory is accessible, specifying its offset in that address map.

```
class my_blk_type extends uvm_reg_block;
   virtual function build();
      this.R1 = my_reg1_type::type_id::create("R1", null, get_full_name());
      this.R1.configure(this,...);
      this.R1.build();
      this.default_map.add_reg(this.R1, `h04, ...);
   endfunction
endclass
```

After a register file has been built, the `build()` method shall call its `map()` method for all address maps where the register file is accessible, specifying its offset in that address map.

```
class my_rf_type extends uvm_reg_regfile;
   virtual function build();
      this.RF1 = my_rf1_type::type_id::create("RF1", null, get_full_name());
      this.RF1.build();
      this.RF1.map(this.default_map, `h200, ...);
   endfunction
endclass
```

After a sub-block has been built, for each address map in that sub-block, the `build()` method shall call the appropriate `uvm_reg_map::add_submap()` method for all address maps where the sub-block address map is accessible, specifying its offset in that upper address map.

```
class my_blk_type extends uvm_reg_block;
   virtual function build();
      this.BLK1.build();
      this.default_map.add_submap(this.BLK1.default_map, `h8000);
   endfunction
endclass
```

5.5.5.4 Coverage Model

A block-level coverage model is defined and instantiated in the block type class. It measures the coverage of the accessed offsets and field values on each instance of that block type. The `uvm_reg_block::sample()` or `uvm_reg_block::sample_values()` methods shall be used to trigger the sampling of a coverage point, based on the data provided as an argument or gathered from the current state of the block type instance. The sampling of the coverage model shall only occur if sampling for the corresponding coverage model has been turned on, as reported by the `uvm_reg_block::get_coverage()` method.

```
class my_blk extends uvm_reg_block;
   covergroup cg_vals;
      ...
   endgroup
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   ...
   virtual function void sample_values();
      super.sample_values();
      if (get_coverage(UVM_CVR_FIELD_VALS))
         cg_vals.sample();
   endfunction
endclass : my_blk
```

All the coverage models that may be included in the block type shall be reported to the `uvm_reg_mem` base class using `uvm_reg_block::build_coverage()` when `super.new()` is called or using the `uvm_reg_block::add_coverage()` method. If no functional coverage models are included in the generated block type, `UVM_NO_COVERAGE` shall be specified. Block-level coverage groups shall only be instantiated in the constructor if the construction of the corresponding coverage model is enabled, as reported by the `uvm_reg_block::has_coverage()` method.

```
class my_blk extends uvm_reg_block;
   covergroup cg_vals;
      ...
   endgroup
   ...
   function new(string name = "my_blk");
      super.new(name, build_coverage(UVM_CVR_FIELD_VALS));
      if (has_coverage(UVM_CVR_FIELD_VALS))
         cg_vals = new();
   endfunction: new
   ...
endclass : my_blk
```

The content, structure, and options of the coverage group is defined by the register model generator and is outside the scope of UVM.

5.5.6 Packaging a Register Model

The generator is free to structure the generated code into packages and files to facilitate compilation or reuse.

The following practices are recommended, but not required:

a) Block types, and all the register, register file, and memory types they require, should be located in separate packages.

b) Register, register file, and memory types shared by more than one block type should be located in separate packages.

c) A header file, with all the required import statements to use the register model, should be generated.

d) A lengthy `build()` method may be split into several, shorter sub-methods. The sub-methods shall be declared local and called by the `build()` method.

5.5.7 Maximum Data Size

By default, the maximum size of fields, registers, and memories is 64 bits. This limitation is implemented via the definition of the `uvm_reg_data_t` type.

```
typedef bit [63:0] uvm_reg_data_t;
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

The `uvm_reg_data_t` type is used in all methods and API that deal with data values to and from the register model. Smaller fields, registers, and memories are intrinsically supported by using the SystemVerilog automatic value extension and truncation.

The maximum data size may be reduced to save memory in large register models. It may also be increased to support larger fields, registers, or memory values. The size of data values may be specified at compile-time by defining the `` `UVM_REG_DATA_WIDTH `` macro.

```
% ... +define+UVM_REG_DATA_WIDTH=256 ...
```

It is recommended register model generator provide a warning message if the maximum data size need to be increased. It is also recommended the register model contain a static initializer check for the required minimum data size and issue a fatal error message when that is not set appropriately:

```
class my_blk extends uvm_reg_block;
   local static bit m_req_data_width = check_data_width(256);
   ...
endclass
```

5.6 Back-door Access

Back-door access to registers and memory locations is an important tool for efficiently verifying their correct operation.

A back-door access can uncover bugs that may be hidden because write and read cycles are performed using the same access path. For example, if the wrong memory is accessed or the data bits are reversed, whatever bug is introduced on the way in (during the write cycle) will be undone on the way out (during the read cycle).

A back-door improves the efficiency of verifying registers and memories since it can access registers and memory locations with little or no simulation time. Later, once the proper operation of the physical interface has been demonstrated, you can use back-door access to completely eliminate the simulation time required to configure the DUT (which can sometimes be a lengthy process).

A back-door access operates by directly accessing the simulation constructs that implement the register or memory model through a hierarchical path within the design hierarchy. The main challenges of implementing a back-door access are the identification and maintenance of that hierarchical path and the nature of the simulation constructs used to implement the register or memory model.

5.6.1 Back-door read/write vs. peek/poke

You can perform back-door access to registers and memory by calling the following read/write methods with their `path` argument as `UVM_BACKDOOR`:

a) `uvm_reg_field::read()` or `uvm_reg_field::write()`
b) `uvm_reg::read()` or `uvm_reg::write()`
c) `uvm_mem::read()` or `uvm_mem::write()`

... or by calling the following peek/poke methods:

d) `uvm_reg_field::peek()` or `uvm_reg_field::poke()`
e) `uvm_reg::peek()` or `uvm_reg::poke()`
f) `uvm_mem::peek()` or `uvm_mem::poke()`

Copyright © 2011 - 2015 Accellera. All rights reserved.

The `peek()` methods return the raw value read using the back-door without modifying the content of the register or memory. Should the register content be modified upon a normal read operation, such as a `clear-on-read` field, it will not be modified. Therefore, reading using `peek()` methods may yield different results than reading through `read()` methods.

The `poke()` methods deposit the specified value directly in the register or memory. Should the register contain non-writable bits or bits that do not reflect the exact value written, such as `read-only` or `write-1-to-clear` fields, they will contain a different value than if the same value had been written through normal means. All field values, regardless of their access mode, will be forced to the poked value. Therefore, writing using `poke()` methods may yield different results than writing through the front-door.

When using the `read()` methods with a back-door access path, the behavior of the register or memory access mimics the same access performed using a front-door access. For example, reading a register containing a `clear-on-read` field will cause the field value to be cleared by poking `0`'s into it.

When using the `write()` method with a back-door access path, the behavior of the register or memory access mimics the same access performed using a front-door access. For example, writing to a `read-only` field using back-door access will cause the field value to be maintained by first peeking its current value then poking it back in instead of the specified value.

5.6.2 Hierarchical HDL Paths

To access a register or memory directly into the design, it is necessary to know how to get at it. The UVM register library can specify arbitrary hierarchical path components for blocks, register files, registers and memories that, when strung together, provide a unique hierarchical reference to a register or memory. For example, a register with a hierarchical path component defined as `X`, inside a block with a hierarchical path component defined as `Y`, inside a block with a hierarchical path component defined as `Z` has a full hierarchical path defined as `Z.Y.X`.

HDL path components are specific to the language used to model the DUT and the structure of the DUT model. They may be individual hierarchical scope names (e.g., `decoder`), partial dot-separated (`.`) hierarchical paths (e.g., `bus_if.decoder`) or empty (i.e., they do not contribute to the overall path). They can also be build-time string expressions, but must be string constants at run-time, where the value of the string must be a constant name or partial path: it cannot be an expression. Each path component must be empty or a valid path component: they cannot start or end with a dot separator (`.`). They need not be valid SystemVerilog path components, as they may be used to refer to hierarchical paths that cross language boundaries. HDL paths terminate at registers and memories.

Multiple HDL paths may be defined for the same block, register file, register or memory abstraction. This indicates the block, register file, register, or memory is duplicated in the model of the DUT. The value of a duplicated register or memory must be kept coherent across all copies.

For example, assuming the following register model hierarchy and HDL path components:

```
Block b1                    "b1"
    Block b2                "b2_a", "b2_b"
        Register r1         "r1"
        Register r2         {"r2_1", "r2_0"}
    Block b3                ""
        Register r3         "r3.z", {"r3_1", "r3_0"}
```

The full hierarchical paths would be as follows:

 Copyright © 2011 - 2015 Accellera. All rights reserved.

Block b1	"b1"
Block b2	"b1.b2_a", "b1.b2_b"
Register r1	"b1.b2_a.r1", "b1.b2_b.r1"
Register r2	{"b1.b2_a.r2_1", "b1.b2_a.r2_0"}, {"b1.b2_b.r2_1", "b1.b2_b.r2_0"}
Block b3	n/a
Register r3	"b1.r3.z", {" b1.r3_1", "b1.r3_0"}

HDL path components are specified using the following methods:

a) `uvm_reg_block::configure()` and `uvm_reg_block::add_hdl_path()`

b) `uvm_reg_file::configure()` and `uvm_reg_file::add_hdl_path()`

c) `uvm_reg::configure()` and `uvm_reg::add_hdl_path_slice()`

d) `uvm_mem::configure()` and `uvm_mem::add_hdl_path_slice()`

The HDL path for a register or memory may be a concatenation of simple names. This is used when the register is implemented or modeled as a concatenation of individual fields variables, or when the memory is implemented or modeled using vertical banking. When specifying a concatenation, the bits may be left unspecified if they are not physically implemented.

For example, the register with the implementation illustrated in Figure 26 has its HDL path component specified using the following concatenation:

```
rg.add_hdl_path_slice("RDY", 15, 1);
rg.add_hdl_path_slice("ID", 6, 6);
rg.add_hdl_path_slice("COUNT", 0, 4);
```

Figure 26—Composite Register Structure

HDL paths are created by concatenating each path component from the root block to the leaf register or memory. However, if a block HDL path component is defined has a root HDL path component, the HDL path component of any blocks above it are ignored.

5.6.3 VPI-based Back-door Access

The UVM register library provides a default back-door access mechanism that uses the HDL path(s) returned by the `uvm_reg::get_full_hdl_path()` and `uvm_reg::get_full_hdl_path()` methods for the default design abstraction. Using standard SystemVerilog VPI routines, it samples or deposits values in the HDL constructs referenced by the resulting hierarchical HDL paths. If the HDL paths are valid hierarchical SystemVerilog variables, including indexing and slicing operators, this should work without any further requirements.

5.6.3.1 Including the DPI C Library

The implementation of the default back-door access mechanism requires the inclusion of some DPI C code. Please refer to the *supplementary* UVM *documentation* provided by your simulator vendor on how to compile and link the UVM C library.

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.6.3.2 Performance Issues with the VPI-based Back-door Access

Enabling the VPI functionality required by the default back-door access mechanism may disable performance optimizations normally done in your simulator. Please refer to the *supplementary* UVM *documentation* provided by your simulator vendor for additional or alternative steps that may be taken to improve the performance of your simulation.

5.6.4 User-defined Back-door Access

Should the DPI-based back-door access prove to be insufficient, a user-defined back-door access can be used instead. A user-defined back-door access is able to use any SystemVerilog constructs or tool-specific utility to access registers and memories. For example, if a memory or register is located in an encrypted model, a user-defined back-door may be used to peek and poke values directly into the encrypted model via a suitable API.

A user-defined register back-door is provided through an extension of the `uvm_reg_backdoor` class. A back-door write operation is implemented in the `uvm_reg_backdoor::write()` virtual method whereas a back-door read operation is implemented in the `uvm_reg_backdoor::read()` virtual method. This back-door access is then associated with a specific register through the `uvm_reg::set_backdoor()` method or with all the registers within a block using the `uvm_reg_block::set_backdoor()` method.

A user-defined memory back-door is provided through an extension of the `uvm_mem_backdoor` class. A back-door write operation is implemented in the `uvm_mem_backdoor::write()` virtual method whereas a back-door read operation is implemented in the `uvm_mem_backdoor::read()` virtual method. This back-door access is then associated with a specific memory through the `uvm_mem::set_backdoor()` method.

User-defined back-door access mechanisms may be defined by the register model generators. They are instantiated and associated with their corresponding block, register, or memory abstraction class in the implementation of their respective `build()` method. User-defined back-door access mechanisms may also be registered with their corresponding block, register, or memory after the register model construction, overriding any previously defined (or default) back-door access mechanisms. In the latter case, they are instantiated and associated with their corresponding block, register, or memory abstraction class in the implementation of the environment's `build()` method.

```
function void
   tb_env::build_phase(uvm_phase phase);
   super.build_phase(phase);
   ...
   begin
      my_mem_backdoor bkdr = new;
      regmodel.mem.set_backdoor(bkdr);
   end
endfunction: build_phase
```

5.6.5 Back-door Access for Protected Memories

The content of memories may be protected using one or more protection schemes. They can vary from simple additional bits providing an Error Correction Code to full encryption of its content.

When performing back-door write operations, it is necessary to correctly protect the memory content to avoid errors when a physical interface subsequently reads these memory locations. It may also be useful or

 Copyright © 2011 - 2015 Accellera. All rights reserved.

necessary to have direct access to the protected form because these bits are created and used entirely within the design, and can only be accessed through back-door access. The back-door is the only way protected values can be checked and protection errors injected.

The `encode()` and `decode()` callback methods located in the `uvm_mem_backdoor_cbs` class are designed to handle such data protection. The `encode()` method is applied on the way in and the `decode()` method is applied on the way out. But, unlike regular callback methods, the decoding is done in the reverse order of registration. This allows multiple layers of data protections to be implemented in the same memory, each modeled using a single callback extension. The order of registration determines the order in which the various layers of protections are applied—then undone.

For example, ECC bits are located in additional memory bits within the same memory location as the data they protect; they must be generated and set for write accesses, and must checked and masked when read.

```
class ecc_protected extends uvm_mem_backdoor_cbs;

   virtual function uvm_reg_data_t encode(uvm_reg_data_t data);
      // Append the ECC bits to the data to write
      data[35:32] = ecc::compute(data[31:0]);
      return data;
   endfunction

   virtual function uvm_reg_data_t decode(uvm_reg_data_t data);
      // Check and mask the ECC bits to the data to write
      if (data[35:32] != ecc::compute(data[31:0])) `uvm_error(...)
      return data[31:0];
   endfunction

endclass
```

Similarly, data written to an encrypted memory must be ciphered during write accesses and deciphered when read.

```
class encrypted_mem extends uvm_mem_backdoor_cbs;

   virtual function uvm_reg_data_t encode(uvm_reg_data_t data);
      return crypt::encrypt(data);
   endfunction

   virtual function uvm_reg_data_t decode(uvm_reg_data_t data);
      return crypt::decrypt(data);
   endfunction

endclass
```

5.6.6 Active Monitoring

The mirrored field values in a register model are updated when the fields are accessed through the register model based on the current mirrored value, the accessed data value, and the access policy of the field. They may also be updated based on observed read and write transactions on the bus interface if the register model is integrated with the bus monitor and explicit monitoring is enabled (see Section 5.9.3). Any changes to the field value performed by the design itself cannot be detected and then mirrored in the register model.

Copyright © 2011 - 2015 Accellera. All rights reserved.

The back-door mechanism can be used to automatically update the mirror value of fields that are modified by the design itself by observing the SystemVerilog constructs which are used to store the field values. When a change of value is detected, the mirrored value can be similarly updated.

Because there is no standard value-change callback VPI or PLI functionality, the automatic update of a field can only be implemented using a user-defined back-door. The active monitoring of a register requires the implementation of the `uvm_reg_backdoor::is_auto_updated()` and `uvm_reg_backdoor::wait_for_change()` methods.

`uvm_reg_backdoor::is_auto_updated()` returns `TRUE` if the specified named field is actively monitored. All actively-monitored fields have their mirror value updated strictly and only through the active mirroring mechanism. Executed or observed transactions are not used to update their mirrored value.

The `uvm_reg_backdoor::wait_for_change()` task must return only when a change in any of the actively-monitored fields is observed. For each actively-monitored register, a thread calls this task to wait for any change in any of the fields in the register. As soon as it returns, their values are sampled and their mirror values updated. The implementation of that method should not simply wait for the active edge of the clock signal used to update the field values in the design; for optimal performance, the implementation of that method should only return when an actual change occurs.

```
class active_monitor_r1 extends uvm_reg_backdoor;

   virtual function bit is_auto_updated(string fld_name);
      case (fld_name)
      "f1": return 1;
      "f2": return 1;
      endcase
   endfunction

   virtual task wait_for_change();
      @($root.tb_top.dut.rf.f1 or $root.tb_top.dut.rf.f2);
   endtask
endclass
```

The active-monitoring thread must be started for each actively-monitored register by invoking the `uvm_reg_backdoor::start_update_thread()` method of its back-door access class once an instance of that back-door access class is created, as shown in the following example:

```
class add_active_monitors extends my_blk;

   virtual function build();
      super.build();
      begin
         active_monitor_r1 am_r1 = new;
         r1.set_backdoor(am_r1);
         am_r1.start_update_thread(r1);
      end
   endfunction
endclass
```

5.7 Special Registers

The UVM register library presumes all registers and memories are average registers and memories, they are accessible at a known, constant, unique physical address(es), their behavior is constant throughout the simulation regardless of the physical interface used to access them, and they contain a single value.

 Copyright © 2011 - 2015 Accellera. All rights reserved.

Designer creativity, the demands of the application, or implementation constraints often require special behaviors be implemented. Special register behavior can be modeled using any number of extension capabilities provided in the UVM register and field abstraction classes. Pre- and post-read/write callback objects, virtual callback methods, user-defined front-doors, and user-defined back-doors may be used to extend the behavior of the base library. And, if all else fails, it is always possible to override virtual methods that are used to access the register content, i.e., `read()`, `write()`, `peek()`, and `poke()`.

5.7.1 Pre-defined Special Registers

The UVM library pre-defines some commonly used special registers. A register model generator is free to provide a library of additional special register models and use them in its generated model.

5.7.1.1 Indirect Indexed Registers

Some registers are not directly accessible via a dedicated address. Indirect access of an array of such registers is accomplished by first writing an "index" register with a value that specifies the array's offset, followed by a read or write of a "data" register to obtain or set the value for the register at that specified offset. The pre-defined `uvm_reg_indirect_data` class models the behavior the "data" register.

A "data" register type is defined by extending the `uvm_reg_indirect_data` register class. The "data" register must not contain any fields. The "index" and indirect register array must be built first, as the "index" registers and the register array are specified when the "data" register is configured using the `uvm_reg_indirect_data::configure()` method. The indirect register array, "index", and "data" registers are added as members of the containing block. However, only the "index" and "data" registers are added to a map in the containing block. The registers in the indirect register array must be not added to the address map in the containing block because they have no dedicated address.

```
class my_blk_type extends uvm_reg_block;
   ind_idx_reg  IND_IDX;
   ind_data_reg IND_DATA;
   ind_reg      INDIRECT_REG[256];

   virtual function build();

      foreach (INDIRECT_REG[i]) begin
         string name = $sformatf("INDIRECT_REG[%0d]",i);
         INDIRECT_REG[i]=
               ind_reg::type_id::create(name,,get_full_name());
         INDIRECT_REG[i].configure(this, null, ...);
         INDIRECT_REG[i].build();
      end

      IND_IDX = ind_idx_reg::type_id::create("IND_IDX",,get_full_name());
      IND_IDX.configure(this, null, ...);
      IND_IDX.build();

      IND_DATA = ind_data_reg::type_id::create("IND_DATA",,get_full_name());
      IND_DATA.configure(IND_IDX, INDIRECT_REG, this, null);
      IND_DATA.build();

      default_map = create_map("", 0, 4, UVM_BIG_ENDIAN);
      default_map.add_reg(IND_IDX, 0);
      default_map.add_reg(IND_DATA, 4);
   endfunction
endclass
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

The registers in the indirect register array cannot be accessed via a back-door access to the "data" register. Back-door access to the register array is provided by performing back-door accesses via the unmapped, indirect register itself.

If a different indirection mechanism is required, a user-defined register extension will be necessary.

5.7.1.2 FIFO (first-in, first-out) Registers

A FIFO register is not a register in the usual sense. It is a FIFO whose push and pop operations are mapped to write and read operations at a specific address. Writing to that address causes the data written to be pushed at the end of the FIFO. Reading from that address returns the data that is currently at the head of the FIFO and pops it. Whether the FIFO is full or empty is usually specified via status bits in another register.

To model a FIFO register, the register type shall be extended from the `uvm_reg_fifo` class. The maximum number of entries in the FIFO and the size of each entry is specified when calling `super.new()`.

```
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
```

Backdoor access to a FIFO register is not allowed.

5.7.2 Unmapped Registers and Memories

By default, the entire register or memory is assumed to be linearly mapped into the address space of the block that instantiates it. Each register or location in a memory thus corresponds to a unique address in the block. However, you can use different addressing mechanisms. For example, you could access a large memory in a limited address space using an indexing mechanism: the desired offset within the memory is written into a register, then the data at that memory offset is read or written by reading or writing another register. This memory is effectively unmapped: it does not appear in the linear address space used to access it. See Section 5.7.1.1.

The number of possible access mechanisms is potentially infinite and only limited by the imagination, requirements, and constraints of designers. To support arbitrary access mechanisms, it is possible to replace the default linearly mapped access mechanism with any user-defined access mechanism.

5.7.2.1 User-defined Front-door Access

User-defined front-door access is made possible by extending the `uvm_reg_frontdoor` class and registering an instance of the class with specific registers or memories using the `uvm_reg::set_frontdoor()` or `uvm_mem::set_frontdoor()` method. The `uvm_reg_frontdoor` is a `uvm_sequence`. For each write or read operation, the register model creates a `uvm_reg_item` object representing the operation, assigns it to the `rw_info` property of registered front-door sequence, and calls its `start` method. Ultimately, the front-door's body task is called, which must be implement to perform the actual operation.

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
class indexed_reg_frontdoor extends uvm_reg_frontdoor;
   local uvm_reg   m_idx_reg;
   local uvm_reg   m_data_reg;
   local bit [7:0] m_addr;

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

   function void configure(uvm_reg idx, uvm_reg data, bit [7:0] addr);
      m_idx  = idx;;
      m_data = data;
      m_addr = addr;
   endfunction: new

   virtual task body(uvm_reg_item rw);
      m_idx_reg.write(status, m_addr, ...);
      if (status != UVM_IS_OK)
         return;
      if (rw.kind == UVM_WRITE)
         m_data.write(rw.status, data, ...);
      else
         m_data.read(rw.status, data, ...);
   endtask

endclass
```

User-defined front-doors are instantiated and associated with their corresponding register or memory abstraction class in the `build()` method of the block or register file that instantiates them or the `build()` phase callback of the environment component where the register model is instantiated and built.

```
virtual function void build();
   foreach TABLE[i] begin
      indexed_reg_frontdoor idx_frtdr = new(INDEX, DATA, i);
          = idx_reg_frontdoor::type_id::create("idx_frtdr",,get_full_name());
      idx_frntdoor.configure(idx_reg, data_reg, i);
      regmodel.TABLE[i].set_frontdoor(idx_frontdoor, default_map, ...);
   end
endfunction: build
```

A user-defined front-door is registered on a per-map basis, affecting the access of a register or memory through a specific physical interface. A different front-door mechanism (or the built-in one) can be used for other physical interfaces. For example, a memory could use the indexed addressing scheme described above for one physical interface but be mapped normally within the address map of another physical interface.

5.7.2.2 Mirroring Unmapped Registers

When using explicit or passive monitoring to update the mirror value in unmapped registers, it will be necessary to override the `uvm_reg::predict()` method of the register(s) used to access the unmapped registers, since the observed transactions will be using the address of those access registers, not the unmapped (unaddressable) registers that are ultimately accessed.

In the case of an indirect register, the `uvm_reg_indirect_data` class extends `predict` for you and serves as an example of how you do this for your custom unmapped registers.

```
function bit uvm_reg_indirect_data::predict (uvm_reg_data_t value, ...);
   if (m_idx.get() >= m_tbl.size()) begin
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

```
      `uvm_error("Index reg > than size of indirect register array")
      return 0;
   end
   return m_tbl[m_idx.get()].predict(value, ...);
endfunction
```

5.7.3 Aliased Registers

Aliased registers are registers that are accessible from multiple addresses in the same address map. They are different from shared registers as the latter are accessible from multiple address maps. Typically, the fields in aliased registers will have different behavior depending on the address used to access them. For example, the fields in a register may be readable and writable when accessed using one address, but read-only when accessed from another.

Modelling aliased registers in UVM involves more than simply mapping the same register at two different addresses. A UVM register model requires each instance of a `uvm_reg` class be mapped to a unique address in an address map. For aliased registers, this requires a register class instance for each address. All this enables using a specific register instance to access the aliased register via a specific address.

For example, the (incomplete) register model shown below models a register aliased at two addresses: `` `h0100 `` and `` `h0200 ``. Each alias is known under a different instance name, `Ra` and `Rb` respectively. To access the aliased register via address `` `h0100 ``, the `Ra` instance would be used.

```
class my_blk extends uvm_reg_block;
   rand my_reg_Ra Ra;
   rand my_reg_Rb Rb;
   virtual function build();
      ...
      default_map.add_reg(Ra, `h0100);
      default_map.add_reg(Rb, `h0200);
   endfunction
endclass
```

Each register instance must be of a register type that models the behavior of the register and field it contains of its corresponding alias. For example, a register that contains a field that is `RW` when accessed via one address, but `RO` when accessed via another would require two register types: one with a `RW` field and another one with a `RW` field, and both using the same field names.

```
class my_reg_Ra extends uvm_reg;
   rand uvm_reg_field F1;
   ...
   virtual function void build();
      F1 = uvm_reg_field::type_id::create("F1");
      F1.configure(this, 8, 0, "RW", 0, 8'h0, 1, 1, 1);
   endfunction
   ...
endclass

class my_reg_Rb extends uvm_reg;
   uvm_reg_field F1;
   ...
   virtual function void build();
      F1 = uvm_reg_field::type_id::create("F1");
      F1.configure(this, 8, 0, "RO", 0, 8'h0, 1, 0, 1);
   endfunction
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   ...
endclass
```

The aliasing functionality must be provided in a third class that links the two register type instances. The aliasing class can make use of the pre-defined register and field callback methods to implement the aliasing functionality. It may also make use of additional APIs or functionality created by the register model generator in the different register types that model each alias of the register. The aliasing class should be based on `uvm_object` to be factory-enabled. The required reference to the various register instance aliases shall be supplied via a `configure()` method.

```
class write_also_to_F extends uvm_reg_cbs;
   local uvm_reg_field m_toF;

   function new(uvm_reg_field toF);
      m_toF = toF;
   endfunction

   virtual function void post_predict(uvm_reg_field  fld,
                                      uvm_reg_data_t value,
                                      uvm_predict_e  kind,
                                      uvm_path_e     path,
                                      uvm_reg_map    map);
      if (kind != UVM_PREDICT_WRITE) return;

      void'(m_toF.predict(value, -1, UVM_PREDICT_WRITE, path, map));
   endfunction

endclass

class alias_RaRb extends uvm_object;
   protected reg_Ra m_Ra;
   protected reg_Rb m_Rb;

   `uvm_object_utils(alias_RaRb)

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

   function void configure(reg_Ra Ra, reg_Rb Rb);
      write_also_to_F F2F;

      m_Ra = Ra;
      m_Rb = Rb;

      F2F = new(Rb.F1);
      uvm_reg_field_cb::add(Ra.F1, F2F);
   endfunction : configure
endclass : alias_RaRb
```

The register file or block containing the various register aliases shall also instantiate the aliasing class in its `build()` method and call the `configure()` method with appropriate arguments.

```
class my_blk extends uvm_reg_block;
   rand my_reg_Ra Ra;
   rand my_reg_Rb Rb;
   ...
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   virtual function build();
      default_map = create_map("", 0, 4, UVM_BIG_ENDIAN);

      Ra = reg_Ra::type_id::create("Ra",,get_full_name());
      Ra.configure(this, null);
      Ra.build();

      Rb = reg_Rb::type_id::create("Rb",,get_full_name());
      Rb.configure(this, null);
      Rb.build();

      default_map.add_reg(Ra, `h0100);
      default_map.add_reg(Rb, `h0200);

      begin
         alias_RaRb RaRb;

         RaRb = alias_RaRb::type_id::create("RaRb",,get_full_name());
         RaRb.configure(Ra, Rb);
      end
   endfunction
endclass
```

There are no pre-defined aliasing classes because the nature of the aliasing is highly variable, not just in how the fields provide different behaviors through the various aliases, but potentially in their layout as well.

5.7.4 Unimplemented Registers

A UVM register model can model registers that are specified, but have not yet been implemented in the DUV. This allows the verification environment and testcases to make use of these registers before they are available.

Because these registers are unimplemented, there is nothing to actually read or write inside the DUT. Since the mirror in a register abstraction class provides a faithful model of the expected behavior of that register, it can be used to provide a read back value. A yet-to-be-implemented register is thus modeled by writing to and reading from the mirror.

An unimplemented register can be modeled by providing a user-defined front- and back-door that access the mirrored value instead of performing bus transactions.

```
class not_yet_implemented_fd extends uvm_reg_frontdoor;
   ...
   virtual task body();
      uvm_reg R;

      $cast(R, rw_info.element);
      if (rw_info.kind == UVM_READ) rw_info.value[0] = R.get();

      R.predict(rw_info.value[0], -1,
                (rw_info.kind == UVM_READ) ?
                   UVM_PREDICT_READ : UVM_PREDICT_WRITE,
                rw_info.path,
                rw_info.map);
   endtask
endclass
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

The user-defined front- and back-door can be registered in the environment where the register model is instantiated. As Register model generators will not have access to environment, they can register the back-door and front-door in the `build()` method of the containing block for the unimplemented register. Additionally, it is recommended to log info/warning messages indicating that the particular register is unimplemented.

```
virtual function void connect_phase(uvm_phase phase);
   ...
   if (get_parent() == null) begin
      not_yet_implemented_fd fd;
      `uvm_warning("NotYetImpl",
                   "Working around yet-to-be-implemented registers");
      fd = new;
      regmodel.R2.set_frontdoor(fd);
   end
endfunction
```

5.7.5 RO and WO Registers Sharing the Same Address

It is possible for a register containing only write-only fields (`WO`, `WOC`, `WOS`, and `WO1`) to share the same address with another register containing only read-only fields (`RO`, `RC`, and `RS`). The fields in each register are unrelated and can have different layouts.

This register structure is modeled by simply mapping both registers at the same address. Only one read-only register and one write-only register may be mapped at the same address. Once mapped, calling the `uvm_reg::read()` method on a write-only register or calling the `uvm_reg::write()` method on a read-only register will cause an error message to be issued, the operation will be aborted, and `UVM_NOT_OK` will be returned as the status. Back-door poke and peek are allowed on read-only and write-only registers respectively.

```
class block_B extends uvm_reg_block;
   rand reg_RO R;
   rand reg_WO W;
   ...
   virtual function void build();
      default_map = create_map("", 0, 4, UVM_BIG_ENDIAN);

      R = reg_RO::type_id::create("R");
      R.configure(this, null, "R_reg");
      R.build();

      W = reg_WO::type_id::create("W");
      W.configure(this, null, "W_reg");
      W.build();

      default_map.add_reg(R, 'h100,  "RO");
      default_map.add_reg(W, 'h100,  "WO");
   endfunction : build
   ...
endclass
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

5.8 Integrating a Register Model in a Verification Environment

Test sequences, whether pre-defined or user-defined ones, need a verification environment in which to execute. The register model needs to be an integral part of that verification environment to be used by the tests to access registers and memories in the DUT.

An environment must have a reference to the register model that corresponds to the DUT it verifies. It is recommended a class property named `regmodel` be used for that purpose. To enable vertical reuse of the environment, it must first check if its register model has been defined by a higher-level environment. If not, it must be allocated using the class factory, explicitly built by calling its `build()` method, then it calls the `uvm_reg_block::lock_model()` method. After creating any sub-block environments, their register models must then be specified by setting their respective `regmodel` class properties. All of this must be implemented in the environment's `build phase` method.

```
class block_env extends uvm_env;
   block_reg_model regmodel;
   subblk_env      subblk;

   virtual function void build_phase(uvm_phase phase);
      if (regmodel == null) begin
         regmodel = block_reg_model::type_id::create("regmodel", this);
         regmodel.build();
         regmodel.lock_model();
      end
      subblk = subblk_env::type_id::create("subblk", this);
      subblk.regmodel = regmodel.subblk;
   endfunction
endclass
```

If HDL paths are used, the root HDL paths must be specified in the environment that instantiates the register model. The value of that root path will depend on the location of the model for the DUT within the complete simulation model.

```
class block_env extends uvm_env;
   block_reg_model regmodel;
   virtual function void build_phase(uvm_phase phase);
      if (regmodel == null) begin
         regmodel = block_reg_model::type_id::create("regmodel", this);
         regmodel.build();
         regmodel.set_hdl_path_root("tb_top.dut");
      end
   endfunction
endclass
```

5.9 Integrating a Register Model

A register model must be integrated with the bus agents that perform and monitor the actual read and write operations. The terms "bus driver", "bus agent", "bus interface" and "bus operations" are used to describe the components, protocol, and interface associated with the execution of read and write operations on the DUT. The integration may be established via a non-bus-based interface and protocol.

The integration with the bus agent must only be done on root blocks. Root blocks model the entire DUT and they are the only ones who have access to and knowledge of the externally-visible address maps. Lower-level register models will translate their read and write operations in terms of read and write operations at the root block level, using root-level addresses and bus protocols.

 Copyright © 2011 - 2015 Accellera. All rights reserved.

To that end, the integration process must be conditional to the register model being a root register model. This is accomplished by checking if the register model has a parent. If not, it is a root model and integration with the bus agent may proceed. All this must be implemented in the environment's `connect` phase.

```
class block_env extends uvm_env;

   block_reg_model regmodel;
   subblk_env      subblk;

   virtual function void connect_phase(uvm_phase phase);
      if (regmodel.get_parent() == null) begin
      // Integrate register model with bus agent
      ...
      end
   endfunction
endclass
```

There are three structural bus agent integration approaches for keeping the register model's mirror values in sync with the DUT: implicit prediction, explicit prediction, and passive.

Implicit prediction only requires the integration of the register model with one or more bus sequencers. Updates to the mirror are predicted automatically (i.e., implicitly) by the register model after the completion of each read, write, peek, or poke operation. This integration is the simplest and quickest, but it will fail to observe bus operations that did not originate from the register model (e.g., by a third-party bus agent) and thus fail to appropriately update the corresponding mirror values.

Explicit prediction requires the register model be integrated with both the bus sequencers and corresponding bus monitors. In this mode, implicit prediction is turned off and all updates to the mirror are predicted externally (i.e., explicitly) by a `uvm_reg_predictor` component, one for each bus interface. The predictor receives the bus operations observed by a connected bus monitor, determines the register being accessed by performing a reverse-lookup using the observed address, and then calls the found register's predict method explicitly to update the mirror. This integration requires more work, but it will observe all bus operations, whether they originated from the register model or a third-party bus agent, and thus appropriately update the corresponding mirror values.

Passive integration only requires the integration of the register model with the bus monitor as described above. All the monitoring of the register operations is performed externally to (i.e., explicitly) the register model. All bus operations, whether they originated from the register model or a third-party bus agent, are observed and thus appropriately reflected in the corresponding mirror values. Because the register model is not integrated with a bus sequencer, it cannot be used to read and write register and memories in the DUT, only to track and verify their current value.

5.9.1 Transaction Adapter

The first step in integrating a register model with a bus agent are the conversion functions between a generic read/write bus operation descriptor, `uvm_reg_bus_op`, used by the register model and the protocol-specific read/write transaction descriptor used by the bus agent.

The transaction adapter is implemented by extending the `uvm_reg_adapter` class and implementing the `reg2bus()` and `bus2reg()` methods. Being a `uvm_object`, the bus adapter must implement a suitable `uvm_object` constructor and use the `` `uvm_object_utils() `` macro to enable it for the class factory.

```
class reg2apb_adapter extends uvm_reg_adapter;
```

Copyright © 2011 - 2015 Accellera. All rights reserved.

```
   `uvm_object_utils(reg2apb_adapter)

   function new(string name = "reg2apb_adapter");
      super.new(name);
   endfunction

   virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
      apb_rw apb = apb_rw::type_id::create("apb_rw");
      apb.kind = (rw.kind == UVM_READ) ? apb_rw::READ : apb_rw::WRITE;
      apb.addr = rw.addr;
      apb.data = rw.data;
      return apb;
   endfunction

   virtual function void bus2reg(uvm_sequence_item bus_item,
                                 ref uvm_reg_bus_op rw);
      apb_rw 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.kind ? UVM_READ : UVM_WRITE;
      rw.addr = apb.addr;
      rw.data = apb.data;
      rw.status = UVM_IS_OK;
   endfunction
endclass
```

If the bus protocol supports byte lane enables (i.e., it is possible to read or write individual bytes in a multi-byte bus), the `supports_byte_enable` class property should be set to `TRUE` in the constructor. Similarly, the `provides_responses` class property should be set to `TRUE` if the bus driver returns responses, e.g., the result of a read operation, in a separate response descriptor:

```
class reg2apb_adapter extends uvm_reg_adapter;
   function new(string name = "");
      super.new(name);
      supports_byte_enables = 0;
      provides_responses    = 1;
   endfunction
endclass
```

Because this transaction adapter is specific to the bus agent, not the register model, it should be provided as part of a UVM-compliant bus UVC.

The transaction adapter is then instantiated in the `connect` phase of the environments corresponding to root register models:

```
class block_env extends uvm_env;
   block_reg_model regmodel;
   subblk_env      subblk;

   virtual function void connect_phase(uvm_phase phase);
      ...
      if (regmodel.get_parent() == null) begin
         reg2apb_adapter reg2apb =
           reg2apb_adapter::type_id::create("reg2apb",,get_full_name());
         ...
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
        end
      endfunction

  endclass
```

5.9.2 Integrating Bus Sequencers

All integration approaches require a register model be configured with one or more bus sequencers. The register model becomes a property of a `uvm_reg_sequence` subtype that executes

a) directly on a bus sequencer, if there is only one bus interface providing access to the DUT registers;

b) as a virtual sequence, if there are one or more bus interfaces providing access to the DUT registers;

c) as a register sequence running on a generic, bus-independent sequencer, which is layered on top of a downstream bus sequencer.

Note—To keep the code examples that follow succinct and focused on register model integration, we do not show obtaining handles via configuration or the resources database, or a priori sequence registration to a specific sequencer.

5.9.2.1 Register Sequence Running on the Bus Sequencer

The simplest approach is to run register sequences directly on the bus sequencer, competing directly with all other "native" bus sequences concurrently running on the bus sequencer. The register sequence will, via the register model, produce bus sequence stimulus using a preconfigured bus adapter. This approach is suitable for when the registers being accessed by the register sequence are accessible via a single bus interface, as shown in Figure 27.

Copyright © 2011 - 2015 Accellera. All rights reserved.

Figure 27—Register Sequence Running Directly on a Bus Sequencer

Implementing this approach is accomplished by registering the bus sequencer and corresponding transaction adapter with the appropriate address map in the register model. The model is registered with the user-defined register sequence and the sequence started on the bus sequencer. As with any other running bus sequence, the register sequence's `is_relevant`, `pre_do`, `mid_do`, and `post_do` methods are called during execution of each bus item generated by the model. To gain exclusive access to the bus, the register sequence may also call `grab` or `lock` to prevent other bus sequences from running.

```
class block_env extends uvm_env;

   block_reg_model regmodel;
   apb_agent apb;

   virtual function void connect_phase(uvm_phase phase);
      if (regmodel.get_parent() == null) begin
         reg2apb_adapter reg2apb =
            reg2apb_adapter::type_id::create("reg2apb",,get_full_name());
         regmodel.APB.set_sequencer(apb.sequencer, reg2apb);
         regmodel.set_auto_predict(1);
      end
      ...
   endfunction
   ...
endclass
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

The above example registers an APB bus sequencer and APB-specific bus adapter with the APB address map defined in top-level register model. If the register model defines only a single map, the map may be referenced via the handle `default_map`.

You define a register sequence by extending `uvm_reg_sequence` and defining the `body()` task to use the `model` property.

```
class my_reg_sequence extends uvm_reg_sequence;
   `uvm_object_utils(my_reg_sequence)

   block_reg_model model;

   virtual task body();
      uvm_status_e status;
      uvm_reg_data_t data;

      model.A.write(status, 'h33, .parent(this));
      if (status == UVM_NOT_OK)
         `uvm_error(...)
      model.A.read(status, data, .parent(this));
      if (data != `h33)
         `uvm_error(...)
   endtask

endclass
```

The `uvm_reg_sequence` class parameterizes its base class. This allows you to splice in any user-defined `uvm_sequence` subtype if needed:

```
class VIP_sequence extends uvm_sequence #(VIP_base_item);
class my_reg_sequence extends uvm_reg_sequence (VIP_sequence);
```

Alternatively, you can promote the parameter to your register sequence, which allows the end-user to choose the `super` class:

```
class my_reg_sequence #(type BASE=uvm_sequence #(uvm_reg_item))
                                        extends uvm_reg_sequence #(BASE);
```

To run the register sequence, assign the sequence `model` property and start it on the bus sequencer:

```
class my_test extends uvm_test;
   block_env env;
   virtual function void run_phase(uvm_phase phase);
      my_reg_sequence seq = my_reg_sequence::type_id::create("seq",this);
      seq.start(env.apb.master);
   endfunction
endclass
```

5.9.2.2 Register Sequence Running as a Virtual Sequence

When the registers in the DUT become accessible via more than one physical bus interface, the same register sequence may instead be started as a virtual sequence as the sequencer used in each write/read call is not directly referenced. The register model routes the operation to the appropriate sequencer based on which map is in effect.

Copyright © 2011 - 2015 Accellera. All rights reserved.

Consider a register model with two registers accessible via different bus interfaces, as shown in Figure 28. As in the previous example in Section 5.9.2.1, the sequence calls write and read on `regA` and `regB` without referring to a map or sequencer.

Note—Write and read calls have an optional map argument, but specifying a map explicitly would limit sequence reuse.

Figure 28—Register Sequence Running as a Virtual Sequence

The only difference between this and running directly on a bus sequencer is more than one sequencer/adapter pair is registered with the register model and the register sequence's `start` method is called without specifying a sequencer to run.

```
class block_env extends uvm_env;

   block_reg_model regmodel;
   apb_agent apb;
   wishbone_agent wsh;

   virtual function void connect_phase(uvm_phase phase);
      if (regmodel.get_parent() == null) begin
         reg2apb_adapter reg2apb =
            reg2apb_adapter::type_id::create("reg2apb",,get_full_name());
         reg2wsh_adapter reg2wsh =
            reg2wsh_adapter::type_id::create("reg2wsh",,get_full_name());
         regmodel.APB.set_sequencer(apb.sequencer, reg2apb);
         regmodel.WSH.set_sequencer(wsh.sequencer, reg2wsh);
         regmodel.set_auto_predict(1);
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
        end
        ...
    endfunction
    ...
endclass
```

A register model having more than one configured interface offers interesting timing possibilities. For example, if two registers are accessible via different busses, their accesses can be concurrent:

```
class my_reg_sequence extends uvm_reg_sequence;

    `uvm_object_utils(my_reg_sequence)

    block_reg_model model;

    virtual task body();
       uvm_status_e status;

        fork
           model.APB.write(status, 'h33, .parent(this));
           model.WSH.read(status, ‘h66, .parent(this));
        join

    endtask

endclass
```

To run the register sequence, register the model and start it without specifying a particular sequencer:

```
class my_test extends uvm_test;

    block_env env;

    virtual function void run_phase(uvm_phase phase);
        my_reg_sequence seq = my_reg_sequence::type_id::create("seq",this);
        seq.start(null);
    endfunction

endclass
```

If you needed to grab a particular sequencer for corner-case testing and were not concerned about creating a dependency on a particular sequencer:

```
grab(regmodel.APB.get_sequencer());
...
ungrab(regmode.APB.get_sequencer());
```

5.9.2.3 Register Sequence Running on a Layered Register Sequencer

An alternative integration mechanism is to connect the register model with a register sequencer, then layer that register sequencer on top of the bus sequencer, as shown in Figure 29. The register operations will "execute" as abstract sequence items on the register sequencer, allowing central, bus-independent control of the register sequences. However, this also prevents register sequences from competing directly with or having control over concurrently executing bus sequences (i.e., via `grab` and `ungrab`), mixing register and bus-specific sequences and sequence item execution within the same sequence, and being notified of bus-specific operations (via `pre_do`, `mid_do`, `post_do`, and `is_relevant`). This process also only works with a single bus interface, as all register operations are funneled through a single register sequence.

Copyright © 2011 - 2015 Accellera. All rights reserved.

Figure 29—Register Sequence Running on a Layered Register Sequencer

In this scheme, you are effectively moving the built-in register-to-bus item conversion and bus item execution from the register model to an external translation sequence, which can be overridden to perform custom address translation or item conversions. The register model sends abstract `uvm_reg_item` descriptors to the register sequencer. It is the responsibility of the translation sequence running on the bus sequencer to get these abstract items and convert them to physical bus items and start them on the bus sequencer. The `uvm_reg_sequence` base class provides this functionality. It parameterizes its base class to enable it to run on bus-specific sequencers.

This is implemented in the `connect` phase by first registering a `uvm_reg_item` sequencer and null adapter with the address map corresponding to the bus being targeted. In a single-map model, the `default_map` is typically used.

You then create an instance of a translation sequence and configure it with the register sequencer handle and bus adapter. The pre-defined layering sequence `uvm_reg_sequence`, properly parameterized and configured, may be used in this step.

Then, in the `run` phase, you start the translation sequence on the bus sequencer.

```
// translation sequence type
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
typedef uvm_reg_sequence #(uvm_sequence #(apb_rw)) reg2apb_seq_t;

class block_env extends uvm_env;

   block_reg_model               regmodel;
   uvm_sequencer#(uvm_reg_item)  reg_seqr;
   apb_agent                     apb;
   reg2apb_seq_t                 reg2apb_seq;

   virtual function void connect_phase(uvm_phase phase);
      if (regmodel.get_parent() == null) begin
         regmodel.default_map.set_sequencer(reg_seqr,null);
         reg2apb_seq = reg2apb_seq_t::type_id::create("reg2apb_seq",,
                                                 get_full_name());
         reg2apb_seq.reg_seqr = reg_seqr;
         reg2apb_seq.adapter =
         reg2apb_adapter::type_id::create("reg2apb",,
                                       get_full_name());
         regmodel.set_auto_predict(1);
      end
   endfunction

   virtual task run();
      reg2apb_seq.start(apb.sequencer);
   endtask

endclass
```

To run a register sequence, you register the model and start it on the register sequencer:

```
class my_test extends uvm_test;

   block_env env;

   virtual function void run_phase(uvm_phase phase);
      my_reg_sequence seq = my_reg_sequence::type_id::create("seq",this);
      seq.start(env.reg_seqr);
   endfunction

endclass
```

5.9.3 Integrating the Register Model with a Bus Monitor

By default, the register model updates its mirror copy of the register values implicitly. Every time a register is read or written through the register model, its mirror value is updated. However, if other agents on the bus interface perform read and write transactions outside the context of the register model, the register model must learn of these bus operations to update its mirror accordingly.

Integration of a bus monitor (see Figure 30) to make predictions based on observed transactions is independent from how bus sequencers are integrated. All previously described bus sequencer integration approaches may employ explicit, bus monitor-based prediction.

Copyright © 2011 - 2015 Accellera. All rights reserved.

Figure 30—Integration with a Bus Monitor

The predictor accepts bus transactions from a connected bus monitor. It uses the preconfigured adapter to obtain the canonical address and data from the bus operation. The map is used to lookup the register object associated with that address. The register's `predict()` method is then called with the observed data to update the mirror value. If the register width is wider than the bus, the predictor will collect multiple observed bus operations before calling `predict()` with the register's full value. As a final step, a generic `uvm_reg_item` descriptor representing the abstract register operation is broadcast to subscribers of its analysis port.

Integration is accomplished by first instantiating a `uvm_reg_predictor` component, parameterized to the bus transaction type, and configuring it with the adapter and address map in the register model that corresponds to the bus being monitored. The `uvm_reg_predictor` component is then connected to the bus monitor's analysis port.

```
class block_env extends uvm_env;

   block_reg_model              regmodel;
   uvm_reg_predictor#(apb_rw)   apb2reg_predictor;
   apb_agent                    apb;

   virtual function void build_phase(uvm_phase phase);
      ...
      apb2reg_predictor = new("apb2reg_predictor", this);
   endfunction
```

 Copyright © 2011 - 2015 Accellera. All rights reserved.

```
    virtual function void connect_phase(uvm_phase phase);

        if (regmodel.get_parent() == null) begin
            reg2apb_adapter reg2apb =
                reg2apb_adapter::type_id::create("reg2apb",,get_full_name());
            ...
            apb2reg_predictor.map = regmodel.APB;
            apb2reg_predictor.adapter = reg2apb;
            regmodel.APB.set_auto_predict(0);
            apb.monitor.ap.connect(apb2reg_predictor.bus_in);
        end
        ...
    endfunction
    ...
endclass
```

When explicit prediction is employed, the implicit prediction must be turned off using `uvm_reg_map::set_auto_predict(0)`.

Note—For register models with a single address map, the name of the address map will be `default_map`.

5.10 Randomizing Field Values

A register model can specify constraints on field values. You can add additional constraints by extending the field, register, register file, or block abstraction class and substituting it in the register model using the factory or by using `randomize() with {}` when randomizing a field, register, register file, or block. When constraining a field value, the class property to be constrained is named `value`. This is *not* the class property that is eventually mirrored or updated and used by the `get()` and `set()` methods; it cannot be used for purposes other than random constraints.

```
ok = regmodel.r1.randomize() with { f1.value <='hF; };
```

Once randomized, the selected field values in a register or block may be automatically uploaded to the DUT by using the `uvm_reg::update()` or `uvm_reg_block::update()` method. This will upload any randomized value that is different from the current mirrored value to the DUT. If you override the `post_randomize()` method of a field abstraction class, you must call `super.post_randomize()` to ensure the randomized value is properly set into the mirror.

You can relax constraints specified in a register model by turning the corresponding constraint block `OFF`.

```
regmodel.r1.consistency.constraint_mode(0);
```

5.11 Pre-defined Sequences

Once a register model has been instantiated in an environment and integrated with the DUT, it is possible to execute any of the predefined register tests sequences to verify the proper operation of the registers and memories in the DUV. It is recommended you start with the simplest test—the hardware reset test—to debug the register model, the environment, the physical transactors, and the DUV to a level where it can be taken through more complicated tests. Some of the predefined test sequences require back-door access be available for registers or memories.

The predefined test sequences in Table 13 are included in the register library. You can combine them in a higher-level virtual sequence to better verify your design. Test sequences are not applied to any block,

Copyright © 2011 - 2015 Accellera. All rights reserved.

register, or memory with the `NO_REG_TESTS` attribute defined. Refer to the UVM *1.2 Class Reference* for more details on each pre-defined test sequence.

Table 13—Pre-defined Test Sequences

Sequence Name	Description	Attributes

`uvm_reg_hw_reset_seq`	Reads all the register in a block and check their value is the specified reset value.	Skip block or register if any of the following attributes are defined:
		`NO_REG_HW_RESET_TEST`
		`NO_REG_TESTS`

`uvm_reg_single_bit_bash_seq`	Sequentially writes `1`'s and `0`'s in each bit of the register, checking it is appropriately set or cleared, based on the field access policy specified for the field containing the target bit.	Skip register if any of the following attributes are defined:
		`NO_REG_BIT_BASH_TEST`
		`NO_REG_TESTS`

`uvm_reg_bit_bash_seq`	Executes the `uvm_reg_single_bit_bash_seq` sequence for all registers in a block and sub-blocks.	Skip block if any of the following attributes are defined:
		`NO_REG_BIT_BASH_TEST`
		`NO_REG_TESTS`

`uvm_reg_single_access_seq`	Requires the back-door be defined for the register. For each address map in which the register is accessible, writes the register then confirms the value was written using the back-door. Subsequently writes a value via the back-door and checks the corresponding value can be read through the address map.	Skip register if any of the following attributes are defined:
		`NO_REG_ACCESS_TEST`
		`NO_REG_TESTS`

`uvm_reg_access_seq`	Executes the `uvm_reg_single_access_seq` sequence for all registers in a block and sub-blocks.	Skip block if any of the following attributes are defined:
		`NO_REG_ACCESS_TEST`
		`NO_REG_TESTS`

`uvm_mem_single_walk_seq`	Write a walking pattern into the memory then checks it can be read back with the expected value.	Skip memory if any of the following attributes are defined:
		`NO_MEM_WALK_TEST`
		`NO_MEM_TESTS`
		`NO_REG_TESTS`

`uvm_mem_walk_seq`	Executes the `uvm_mem_single_walk_seq` sequence for all memories in a block and sub-blocks.	Skip block if any of the following attributes are defined:
		`NO_MEM_WALK_TEST`
		`NO_MEM_TESTS`
		`NO_REG_TESTS`

`uvm_mem_single_access_seq`	Requires the back-door be defined the memory. For each address map in which the memory is accessible, writes the memory locations for each memory then confirms the value was written using the back-door. Subsequently writes a value via the back-door and checks the corresponding value can be read through the address map.	Skip memory if any of the following attributes are defined:
		`NO_MEM_ACCESS_TEST`
		`NO_MEM_TESTS`
		`NO_REG_TESTS`

 Copyright © 2011 - 2015 Accellera. All rights reserved.

Table 13—Pre-defined Test Sequences (Continued)

Sequence Name	Description	Attributes

`uvm_mem_access_seq`	Executes the `uvm_mem_single_access_seq` sequence for all memories in a block and sub-blocks.	Skip block if any of the following attributes are defined:
		`NO_MEM_ACCESS_TEST`
		`NO_MEM_TESTS`
		`NO_REG_TESTS`

`uvm_reg_shared_access_seq`	Requires the register be mapped in multiple address maps. For each address map in which the register is accessible, writes the register via one map then confirms the value was written by reading it from all other address maps.	Skip register if any of the following attributes are defined:
		`NO_SHARED_ACCESS_TEST`
		`NO_REG_TESTS`

`uvm_mem_shared_access_seq`	Requires the memory be mapped in multiple address maps. For each address map in which the memory is accessible, writes each memory location via one map then confirms the value was written by reading it from all other address maps.	Skip memory if any of the following attributes are defined:
		`NO_SHARED_ACCESS_TEST`
		`NO_MEM_TESTS`
		`NO_REG_TESTS`

`uvm_reg_mem_shared_access_seq`	Executes the `uvm_reg_shared_access_seq` sequence for all registers in a block and sub-blocks. Executes the `uvm_mem_shared_access_seq` sequence for all memories in a block and sub-blocks.	Skip block if any of the following attributes are defined:
		`NO_SHARED_ACCESS_TEST`
		`NO_MEM_TESTS`
		`NO_REG_TESTS`

`uvm_reg_mem_built_in_seq`	Execute all the selected predefined block-level sequences. By default, all pre-defined block-level sequences are selected.	Applies attributes governing each predefined sequence, as defined above.
`uvm_reg_mem_hdl_paths_seq`	Verify the HDL path(s) specified for registers and memories are valid.	Skip register or memory if no HDL path(s) have been specified.

Copyright © 2011 - 2015 Accellera. All rights reserved.